VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Určení optimální trajektorie pomocí metod umělé inteligence

Determining the Optimal Trajectory using Artificial Intelligence Methods

Jiří Gongol

Bakalářská práce

Vedoucí práce: Ing. Pavel Kodytek

Ostrava, 2021

Poděkování:

Mé poděkování patří panu **Ing. Pavlu Kodytkovi** za odborné vedení, ochotu, trpělivost a cenné rady, které mi v průběhu zpracování mé bakalářské práce věnoval.

Dále bych chtěl poděkovat mé rodině za oporu a zázemí, které mi bylo poskytnuto během studia.

Abstrakt

Tato bakalářská práce pojednává o metodách umělé inteligence využívaných pro hledaní nejkratší trajektorie. Pro řešení této problematiky je využito algoritmů Breadth-first search, Depth-first search a A* algoritmus, ale nejsou opomenuty ani další metody umělé inteligence či rozbor funkce jiných efektivních algoritmů. Výstupem práce je funkční aplikace navržena v programovacím prostředí LabVIEW, která vizuálně simuluje úlohu robota v bludišti, jehož úkolem je najít cestu k cíli. Robota lze ovládat jak automaticky podle zvoleného algoritmu, tak manuálně prostřednictvím tlačítek šipek. Součástí práce je porovnání efektivnosti algoritmů na základě zjištěných výsledků pro určité mapy bludiště. Navržený systém je výhradně softwarového charakteru a je využitelný na akademické půdě jako demonstrace průběhu algoritmů nejkratší trajektorie. Výsledná aplikace je rozšířitelná, škálovatelná, doplnitelná a splňuje standardy NI.

Klíčová slova:

Umělá inteligence, nejkratší trajektorie, algoritmy, Breadth-first search, Depth-first search, A* algoritmus, LabVIEW

Abstract

This bachelor's thesis deals with methods of artificial intelligence used to find the shortest trajectory. Breadth-first search, Depth-first search and A* algorithm are used to solve this issue, but other methods of artificial intelligence and analysis of function of other effective algorithms are not forgotten. Output of this thesis is functional application designed in programming environment LabVIEW program. Aplication visually simulates task of a robot in a labyrinth, whose goal is to reach the treasure. Robot can be controlled automatically through chosen algorithm, but also manually with arrow buttons. Comparison of algorithms effectivity based on discovered results for certain maps of the labyrinthm is part of this thesis. Projected system is solely software based and it could be applied on academical ground as a demonstration of progress of the shortest path algorithms. The final application is expandable, scalable, supplementable and it fulfils standards of NI.

Keywords:

Artificial intelligence, shortest trajectory, algorithms, Breadth-first search, Depth-first search, A* algorithm, LabVIEW

Obsah

Seznam zkratek a symbolů

AI	Artificial intelligence
LLE	Locally linear embedding
HARP	Hierarchical representation learning for networks
SVD	Singular value decomposition
FGE	Fast geometric ensembling
SDNE	Structural deep network embedding
DNGR	Deep neural networks for learning graph representations
PPMI	Positive pointwise mutual information
GCNs	Graph convolutional networks
DFS	Depth-first search
BFS	Breadth-first search
B&B	Branch and bound
A*	A-star algoritmus
E	East
S	South
W	West
N	North
GUI	Graphical user interface
VI	Virtual Instruments
FIFO	First-In First-Out

Seznam ilustrací

Seznam tabulek

1 Úvod

Určení optimální trajektorie je problematika, se kterou se většina lidí potýká téměř každý den. Všichni chceme při plánování našich cest zvolit tu nejkratší, tedy optimální cestu a ušetřit tím čas nebo dokonce i finance. Například, když plánujeme výlet do hor a chceme navštívit určité vrcholy, které jsou od sebe různě daleko, tak se snažíme naši trasu naplánovat tak, abychom si ji zbytečně neprodlužovali, a šli nejkratší trajektorií. Ovšem v tomto případě přepokládáme, že využíváme turisticky značené trasy a neženeme se za vrcholy tzv. po vzdušné čáře, což by nám v některých případech ani nemuselo pomoci a vzhledem k obtížnosti terénu bychom tak čas ani neušetřili.

Cílem této bakalářské práce je rozebrání a následná implementace některých metod umělé inteligence, které mohou vést k řešení této problematiky. K úspěšnému řešení této úlohy existuje široká škála algoritmů. V této práci představuji ty nejznámější a nejpoužívanější algoritmy, avšak podrobněji se budu věnovat pouze těmto algoritmům: Depth-first search, Breadth-first search a A* algoritmu. První dva zmíněné patří k těm nejzákladnějším algoritmům a jsou základním stavebním kamenem pro ty pokročilejší a efektivnější algoritmy, jako je například A* algoritmus, který je v mnoha případech stále považován za nejlepší řešení.

Umělá inteligence neboli Artificial Intelligence (AI) je v dnešní době čím dál populárnějším tématem a oblast jejího využití, která je již velice rozsáhlá, se každou chvílí rozšiřuje. Umělá inteligenci využívá široké spektrum různých metod pro reprezentaci složitých systémů, některé tyto metody jsou zmíněny v kapitole 2.

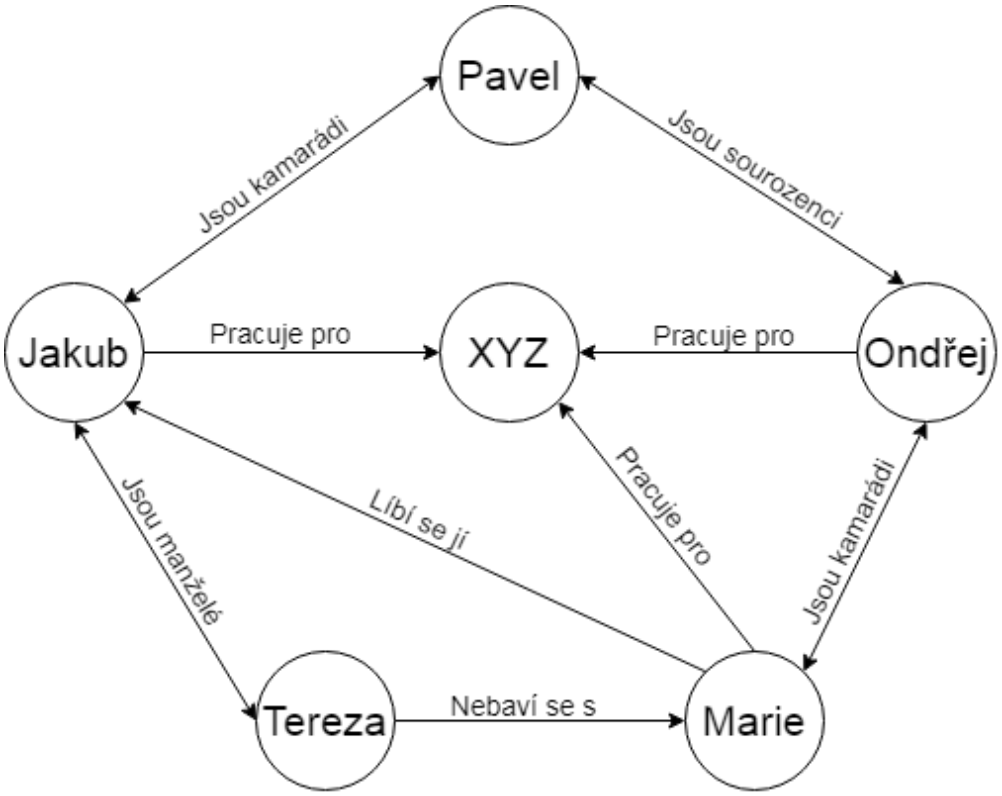
V praktické části zmíněné algoritmy implementuji v grafickém programovacím prostředí LabVIEW na konkrétní úloze, která je definována v Definice úlohy. Programovacímu prostředí LabVIEW věnuji jednu samostatnou kapitolu, ve které rozeberu princip, jak LabVIEW pracuje a jak se v něm vykonává samotný kód. Správnou funkčnost a variabilnost implementovaných algoritmů testuji na více typů map bludiště, aby výsledky byly důvěryhodné. Následně vyhodnotím a porovnám úspěšnost a efektivnost těchto algoritmů na základě zjištěných výsledků.

2 Metodika reprezentace složitých systémů

Mnoho složitých systémů lze reprezentovat pomocí grafů, což nám umožní efektivní ukládání a přístup k relačním znalostem o interaktivních entitách. Entitu může představovat libovolný objekt reálného světa, který je zachycen v datovém modelu. Entita musí být jednoznačně rozlišitelná od ostatních entit a existovat nezávisle na nich. [1]

Jednotlivce ve složitých systémech prezentují uzly v síťových grafech a vztahy mezi jednotlivci jsou reprezentovány jako hrany. Jednoduchý příklad takového grafu je na obrázku 1. Existuje mnoho forem hran: vážené nebo neváženě, orientované nebo neorientované, předepsané nebo nepředepsané. Podle propojení uzlů lze sítě také rozdělit na propojené sítě (mezi libovolnými páry uzlů existuje alespoň jedna cesta) a odpojené sítě (existuje alespoň jedna dvojice uzlů, mezi nimiž není cesta). [2], [3]

Graf odráží nejen strukturované informace, ale také hraje klíčovou roli v moderních metodách strojového učení. Mnoho metod strojového učení používá strukturovaná data grafů jako informace o vlastnostech k předpovídání nebo objevování nových vzorů. Typické aplikace analýzy pomocí grafů zahrnují: vizualizaci struktury, detekci komunity, klasifikaci uzlů a predikci propojení. [4], [5]



Obrázek 1: Síťový graf prezentující vztahy mezi jednotlivci

2.1 Detekce komunity

Cílem detekce komunity je najít síťové podstruktury s těsnými vnitřními propojeními a řídkými externími propojeními. Navíc podle pozorovaných vztahů připojení lze štítky uzlů použít k nalezení společné množiny uzlů. V sociálních sítích mohou štítky uzlů představovat zájmy nebo přesvědčení;

v citačních sítích mohou štítky uzlů představovat témata dokumentu nebo klíčová slova; v biologických sítích mohou štítky uzlů představovat jednotlivé funkce. Tradiční metody detekce komunity lze rozdělit do dvou kategorií. Jednou z nich je optimalizace funkce modularity nebo hustoty modularity. Tento druh metody je založen hlavně na více objektivních metodách evoluční optimalizace a metodách integrace komunity. Kvůli omezení rozlišení funkce modularity a funkce hustoty modularity může tento druh metody ztratit důležité informace o topologii sítě, i když může získat dobré výsledky konvergence. [6], [7]

Druhá je založena na struktuře topologie sítě. Například metody založené na spektrální analýze a metody založené na náhodném procházení. Tento druh metody určuje hlavně podobnost mezi uzly pro detekci komunity. [8]

2.2 Učení grafové reprezentace

Učení grafové reprezentace je v posledních letech velmi rozsáhlým výzkumným směrem v oblasti těžby grafů. Každý uzel ve struktuře grafu je reprezentován nízko dimenzionálním a hustým vektorem (vektor, jenž má většinu hodnot nenulových), tj. proces kódování uzlů v grafu. Toto se mnohdy označuje jako vkládání grafů nebo vkládání uzlů. Jedná se o důležitý aplikační směr metody grafové reprezentace pro zakódování vysoko dimenzionálních neeuklidovských informací o struktuře grafů do vektoru funkcí a další objevování struktury komunity. [9]

2.2.1 Metody vkládání uzlů

Metody vkládání uzlů lze rozdělit do tří hlavních kategorií:

- metody založené na faktorizaci
- metody založené na náhodném procházení (random walk)
- metody založené na hlubokém učení (deep learning)

[10]

Metody založené na faktorizaci:

Metody založené na faktorizaci představují spojení mezi uzly ve formě matice. Maticová faktorizace se používá k získání vložené reprezentace uzlů. Z této kategorie zmíním následující 3 metody:

Metoda Locally Linear Embedding (LLE) předpokládá, že každý uzel je lineární kombinací sousedních uzlů ve vloženém prostoru. [11]

Metoda Laplacian Eigenmaps udržuje dva uzly těsněji uložené, když je váha hran vyšší. Metoda používá funkci kvadratické penalizace pro vzdálenost mezi vložením. Proto při zachování podobnosti uzlu jsou informace o rozdílech často zničeny. Vložení Cauchyho grafu tento problém řeší změnou vzorce kvadratické funkce. Cauchyho graf je jedním z mnoha spojitých pravděpodobnostních rozdělení. [11], [12]

Metoda Hope používá k získání matic podobných uzlů pro udržení aproximace vysokého řádu singulární dekompozici hodnot (SVD), což je rozklad matice na maticový součin unitárních matic sloupcových a řádkových indexů a nulové matice, která má na hlavní diagonále singulární hodnoty matice. [13]

Metody založené na náhodném procházení:

K popisu jednotlivců pohybujících se nepředvídatelnými způsoby se často používají modely náhodných procházek. Metody náhodného procházení jsou zvláště užitečné, když je velikost grafu příliš velká na to, aby byla pojata topologická strukturu celého grafu. DeepWalk je nejstarší metoda vkládání uzlů založená na náhodném procházení. Náhodné procházení uzlů v metodě DeepWalk je založeno na rovnoměrném rozložení pravděpodobnosti mezi všemi hodnotami náhodné veličiny. [14]

Hierarchical Representation Learning for Networks (HARP) funguje tak, že najde menší graf, který se blíží globální struktuře jeho vstupu. Tento zjednodušený graf se používá k osvojení sady počátečních reprezentací, které slouží jako dobrá inicializace pro učení reprezentací v původním podrobném grafu. Kombinace HARP s metodou Deepwalk na základě náhodného procházení může získat lepší výsledek vložení grafu. [15]

Metoda Fast geometric ensembling (FGE) je metoda založená na modelu otevřené sítě toku, která se používá k odhalení podkladové struktury toku a skrytého metrického prostoru různých strategií náhodného procházení v síti. Pomáhá najít nové potenciální aplikace při vkládání. Výše uvedené metody mohou dobře odrážet informace o struktuře sítě. Ale při dalším objevování struktury komunity některých připojených sítí je snadné na jejich okraji objevit fuzzy (nejasný) stav. [16]

Metody založené na hlubokém učení:

S dalším výzkumem hlubokého učení bylo navrženo velké množství metod hluboké neuronové sítě pro grafiku. Strukturální hluboké síťové vkládání (SDNE) používá k udržení blízkosti sítí prvního a druhého řádu hluboký autoencoder, což je typ neuronové sítě používaný k učení efektivního kódování dat bez dozoru. Model se skládá ze dvou částí: bez dozoru a pod dohledem. První zahrnuje autoencoder k vyhledání vloženého uzlu, který může rekonstruovat jeho okolí. Druhá část je založena na mapování funkcí Laplacian, což je diferenciální operátor ve vektorové analýze, který je definovan jako divergence gradientu daného skalárního, nebo obecně tenzorového pole. Pokud jsou podobné uzly ve vloženém prostoru od sebe daleko, mapování prvků bude znevýhodněno. [17], [18]

Deep neural networks for learning graph representations (DNGR) rovněž pro získávání výsledků ze sítě využívá neuronovou síť typu autoencoder. Cílem autoencoderu je naučit se reprezentaci neboli kódování sady dat, obvykle pro redukci rozměrů, pomocí trénování sítě, aby ignorovala „šum“ signálu. Model se skládá ze tří částí: náhodné procházení k získávání sekvence uzlů, generování matice pravděpodobnosti společného výskytu a transformace na matici pozitivních bodových vzájemných informací (PPMI). [19]

Graph Convolutional Networks (GCNs) iterativně agreguje vkládání sousedních uzlů. K určení nového vložení vždy využívá již získané vložení a jeho funkci, která byla použita v předchozí iteraci. Souhrnné vkládání lokálních sousedů umožňuje škálování. Více iterací umožňuje naučit se vkládat uzly k popisu globálních sousedství. [20]

Kromě výše uvedených tří typických kategorií metod vkládání uzlů existují i jiné metody, z nich nejčastější je metoda LINE. Tato metoda definuje dvě funkce pro aproximaci prvního a druhého řádu a minimalizuje kombinaci těchto dvou funkcí. Funkce sousednosti prvního řádu je podobná rozkladu grafů. Jejím účelem je snaha udržet vloženou matici sousednosti a bodový produkt nablízku. Metoda LINE se od výše uvedených tří kategorií liší v tom, že definuje dvě společné rozdělení pravděpodobnosti pro každou dvojici uzlů, jedno pomocí matice sousednosti a druhé pomocí vložení. [21]

3 Rozbor problematiky nejkratší trajektorie

Problém nejkratší cesty má velký význam pro skutečný svět v problematice návrhu sítě, dopravě a telekomunikaci. Deterministická verze problému je snadno vyřešitelná. Ve skutečném světě se však často setkáváme s nejistotou a je třeba ji řešit.

V neznámém prostředí je nalezení nejkratší cesty pro přesun z jednoho místa na druhé náročné. Tato místa lze reprezentovat pomocí uzlů v grafu. Propojený graf lze použít k řešení mnoha praktických aplikací, jako je návrh čipů, spolehlivost sítě, plánování dopravy a klastrová analýza. [22], [23] Existuje mnoho aplikací reálného světa, které jsou založeny na konceptu binárních stromů. O stromu se říká, že je binární, pokud má každý uzel nejvýše dva potomky jako levé a pravé. [24] Některé aplikace, jako je optimální cesta, lze odvodit pomocí algoritmu známého jako problém obchodního cestujícího. [25]

V rámci této bakalářské však problém obchodního cestujícího pouze zmíním a rozbor algoritmů pro hledání optimální trajektorie provedu na zjednodušené úloze, ve které je nejkratší trajektorie patrná na první pohled, a tedy výsledky algoritmů lze snadno ověřit.

3.1 Problém obchodního cestujícího

Problém obchodního cestujícího si vyžádal velkou pozornost matematiků a počítačových vědců, protože je snadno popsatelný, avšak obtížně řešitelný.

Problém obchodního cestujícího si klade následující otázku: „Vzhledem k seznamu měst a vzdálenostem mezi každou dvojicí měst, jaká je nejkratší možná trasa, která navštíví každé město jen jednou, a přitom se vždy vrací do původního města?“ Jedná se o tzv. NP – problém neboli nedeterministický polynomiální problém v kombinatorické optimalizaci. Tento problém a jeho řešení je velice důležité v teoretické informatice a operačním výzkumu. [25]

3.2 Problém nejkratší cesty

Problém nejkratší cesty se snaží najít nejkratší trasu z počátečního bodu do konečného cíle. Obecně platí, že pro představení problému nejkratší cesty používáme grafy. Graf je matematický abstraktní objekt, který obsahuje sady uzlů neboli vrcholů a hran. Hrany spojují dvojice vrcholů. Podél okrajů grafu je možné procházet pohybem z jednoho vrcholu do dalších vrcholů. Podle toho, zda je možné procházet po okrajích oběma směry nebo pouze jedním směrem, se určuje, zda je graf orientovaný nebo neorientovaný. Kromě toho se délky hran často nazývají váhy. Váhy se obvykle používají pro výpočet nejkratší cesty z jednoho bodu do druhého. V reálném světě je možné aplikovat teorii grafů na různé typy scénářů. Například pro představení mapy můžeme použít graf, kde vrcholy představují města a hrany představují trasy, které spojují města. Pokud jsou cesty jednosměrné, bude graf orientovaný, v opačném případě bude neorientovaný. [26]

4 Rozbor algoritmů nejkratší cesty

Existují různé typy algoritmů, které řeší problém s nejkratší cestou. Některé jsou velice efektivní, jiné méně, ale vycházelo se právě z jejich základu pro vytvoření těch nejefektivnějších. Názvy nejznámějších konvenčních algoritmů viz níže, z nichž tučně vyznačené proberu v následujících kapitolách. [27]

- **Depth-First search** (Hledání do hloubky)
- **Breadth-First search** (Prohledávání do šířky)
- **A* search algoritmus**
- **Branch and Bound**
- Hill climbing (Gradientní algoritmus)
- Beam search (Paprskové prohledávání)
- Dijkstrův algoritmus
- Floyd-Warshall algoritmus
- Bellman-Fordův algoritmus

4.1 Rozdělení algoritmů podle závislosti

Výše vyjmenované algoritmy, lze rozdělit na nezávislé a závislé podle tabulky 1. Do první skupiny patří algoritmy, které dokážou fungovat nezávisle na tzv. vážených grafech, ve kterých má každá hrana přidělenou informaci o její váze. Vyhledávání u této skupiny algoritmů není optimální, tedy neexistuje záruka, že najdou nejlepší řešení. V druhé skupině algoritmů je jejich funkčnost závislá na vážených grafech, tedy potřebuji znát váhy jednotlivých hran grafu. Jedním z důvodů, proč potřebují znát informace o váze je ten, že se snaží najít optimální řešení, respektive cestu s nejnižšími náklady, tudíž kalkulují s tímto údajem. Skupina závislých algoritmů se dále dělí na algoritmy, které dokážou pracovat s grafy, ve kterých jsou některé váhy záporná čísla. Tuto vlastnost mají: Floyd-Warshall algoritmus a Bellman-Fordův algoritmus. [27]

Tabulka 1: Rozdělení algoritmů podle závislosti na vážených grafech [27]

Nezávislé	Závislé
Depth-first search	Dijkstrův algoritmus
Breadth-first search	A* search algoritmus
Hill climbing	Floyd-Warshall algoritmus
Beam search	Bellman-Fordův algoritmus

4.2 Depth-first search

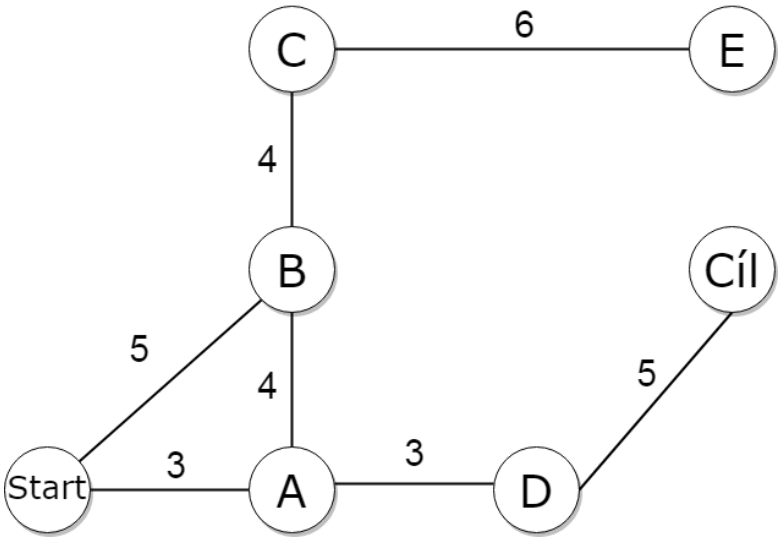
Depth-first search (DFS) neboli hledání do hloubky je algoritmus, který slouží k procházení nebo prohledávání stromových nebo grafových datových struktur. Používá se taktéž k zjišťování počtu komponent topologického uspořádání a detekování cyklů daného grafu. Patří ke skupině těch nejzákladnějších algoritmů. DFS algoritmus vždy expanduje prvního potomka vrcholu, dokud je

to možné, než dorazí na samotný konec větve. Jakmile narazí na vrchol, z kterého není možné pokračovat dále, respektive na vrchol, který nemá žádné potomky, tak se vrací metodou backtracking neboli zpětným vyhledáváním na začátek k vrcholu, kde bylo učiněno poslední rozhodnutí. [27], [28]

4.2.1 Ukázka DFS na příkladu

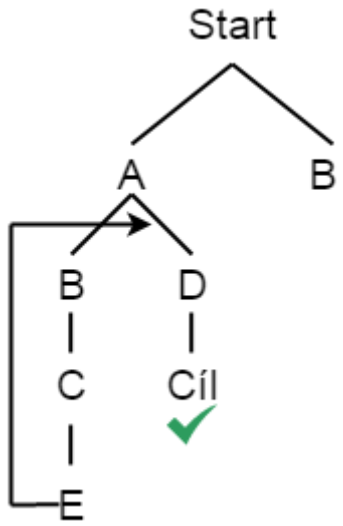
Pro aplikaci a vysvětlení funkce DFS algoritmu použijeme ilustraci jednoduché mapky, která je na obrázku 2. Na mapě je pouze pár důležitých bodů a cestu grafem ze startovací pozice do cíle lze nalézt pouhýma očima. To nám umožní ověření správné funkčnosti algoritmu.

Je třeba si uvědomit, že hledání v těchto metodách není o mapě, ta slouží pouze pro ilustraci a názornost. Nýbrž hledání je o výběru možností, které děláme, když se snažíme udělat rozhodnutí. V tomto případě se jedná o možnosti, které děláme při prozkoumávání mapy. [28]



Obrázek 2: Jednoduchá ilustrace mapky [28]

V této úloze budeme předpokládat, že začínáme v bodě Start a chceme se dostat do bodu Cíl. Průběh algoritmu popisuji podle diagramu průběhu na obrázku 3. Algoritmus začíná v bodě Start a má dvě možnosti postupovat do vrcholu A nebo B. Podle zavedené konvence zpravidla vždy expanduje na potomka, který je vlevo, tedy v tomto případě A. Dále opět dle konvence postupuje přes B a C do vrcholu E, který nemá už žádného následníka, tudíž nemůže pokračovat dále. V tomto případě využije metodu backtracking tedy zpětné vyhledávání a vrátí se do vrcholu, kde učinil poslední rozhodnutí, tedy vrcholu A, a poté expanduje druhého potomka, v tomto případě vrchol D. Z vrcholu D už je pouze jedna možnost, a to sice vrchol Cíl a zde se může algoritmus zastavit, protože splnil svůj úkol. [28]



Obrázek 3: Diagram průběhu DFS algoritmu [28]

4.3 Breadth-first search

Breadth-First search (BFS) neboli prohledávání do šířky je jedním ze základních vyhledávacích algoritmů, slouží pro procházení nebo prohledávání stromových nebo grafových datových struktur. BFS začíná postupovat od kořene stromu nebo od libovolného uzlu grafu a zkoumá všechny sousední uzly v současné hloubce, než přejde k uzlům v další úrovni. Algoritmus si je možné v grafu představit jako šíření ohně. V nulovém kroku je v ohni pouze zdroj, tedy vrchol Start. V každém dalším kroku se oheň hořící v každém vrcholu rozšíří na všechny jeho sousedy. V jedné iteraci algoritmu se „ohnivý kruh“ rozšíří do šířky o jednu jednotku, odtud název algoritmu. [27]

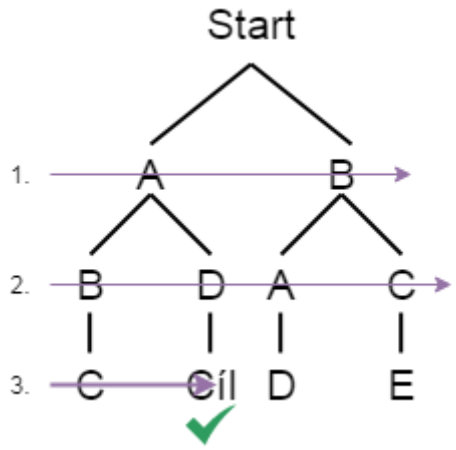
BFS používá opačnou strategii než DFS algoritmus, který místo toho vždy prozkoumá větev uzlu, co nejvíce do hloubky, dokud je to možné, než bude nucen ustoupit a rozšířit další uzly.

BFS a jeho aplikace při hledání propojených komponent grafů vynalezl v roce 1945 Konrad Zuse ve své zamítnuté disertační práci o programovacím jazyce Plankalkül. Práce byla později v roce 1972 zveřejněna. [27]

4.3.1 Ukázka BFS na příkladu

Pro aplikaci a vysvětlení funkce BFS algoritmu, použijeme rovněž stejnou ilustraci jednoduché mapky, která se nachází výše na Obrázek 2.

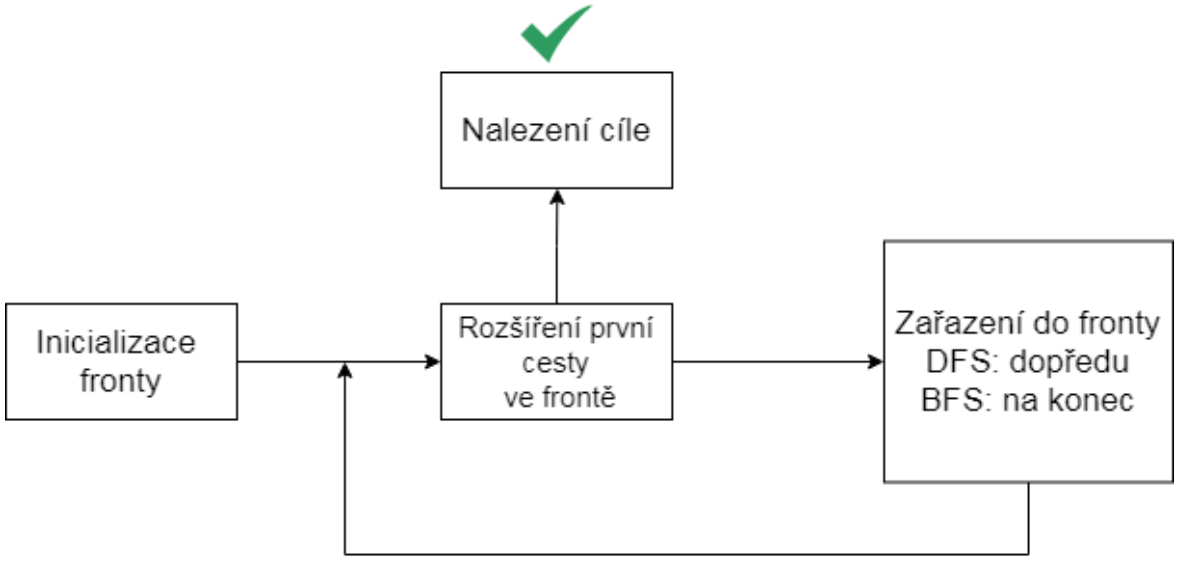
V této úloze budeme opět předpokládat, že začínáme v bodě Start a chceme se dostat do bodu Cíl. Algoritmus BFS začíná postupovat od kořene stromu, tedy bodu Start tak, že zkoumá všechny sousední uzly právě bodu Start v současné hloubce 1, viz obrázek 4. V těchto diagramech zpravidla postupujeme shora dolu a zleva doprava. V úrovni 2 zkoumá všechny sousedy bodů A, B, tedy body B, D, A, C, a to v tomto pořadí, respektive zleva doprava. V třetí iteraci postupně zkoumá všechny sousedy bodů B, D, A, C. Jakmile narazí na náš bod Cíl, tak se zastaví, viz zelená parafa níže v obrázku 4. [28]



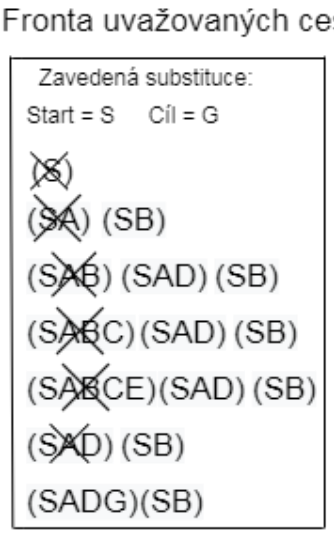
Obrázek 4: Diagram průběhu BFS algoritmu [28]

4.4 Implementace algoritmů a rozbor funkce

Pro vysvětlení funkcí a rozdílu mezi DFS a BFS použiji diagram, viz obrázek 5. Nejdříve zavedu pořadník v podobě fronty, kde budou možné cesty neboli vrcholy, o kterých uvažujeme, viz obrázek 6. V prvním kroku proběhne inicializace fronty, a tedy algoritmus začíná ve vrcholu Start, respektive podle zavedené substituce na obrázku 6 začíná ve vrcholu S. Dále expanduje první cestu ve frontě, když by náhodou při prvním rozšíření nalezl cíl, tak se algoritmus může zastavit, v opačném případě zařadí uvažovanou cestu do fronty a vrací se ke kroku rozšíření první cesty ve frontě. Podle zavedené konvence rozšiřuje nejdříve vrcholy, které leží vlevo, a řadí je dle abecedy, tudíž nejdříve z vrcholu (S) expanduje cestu do (SA). Cestu, kterou expanduje, ve frontě „přeškrtne“, aby věděl, kudy postupuje. Algoritmus takto funguje dokola, dokud nenajde cíl. Tento diagram s malou úpravou platí pro oba algoritmy DFS i BFS. Rozdíl je ten, že v DFS zařazuje uvažované cesty dopředu fronty, naproti tomu v BFS je zařazuje na konec fronty. [28]



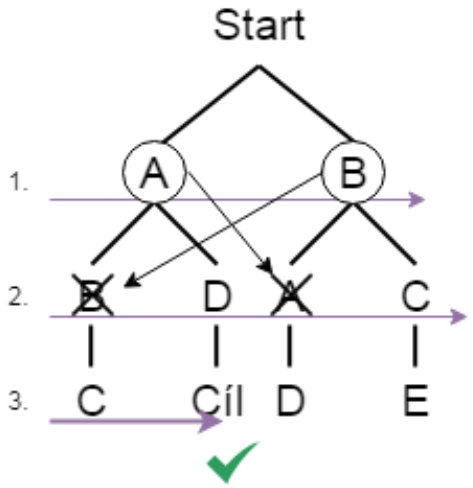
Obrázek 5: Diagram průběhu DFS a BFS algoritmu pro implementaci [28]



Obrázek 6: Fronta uvažovaných cest pro DFS [28]

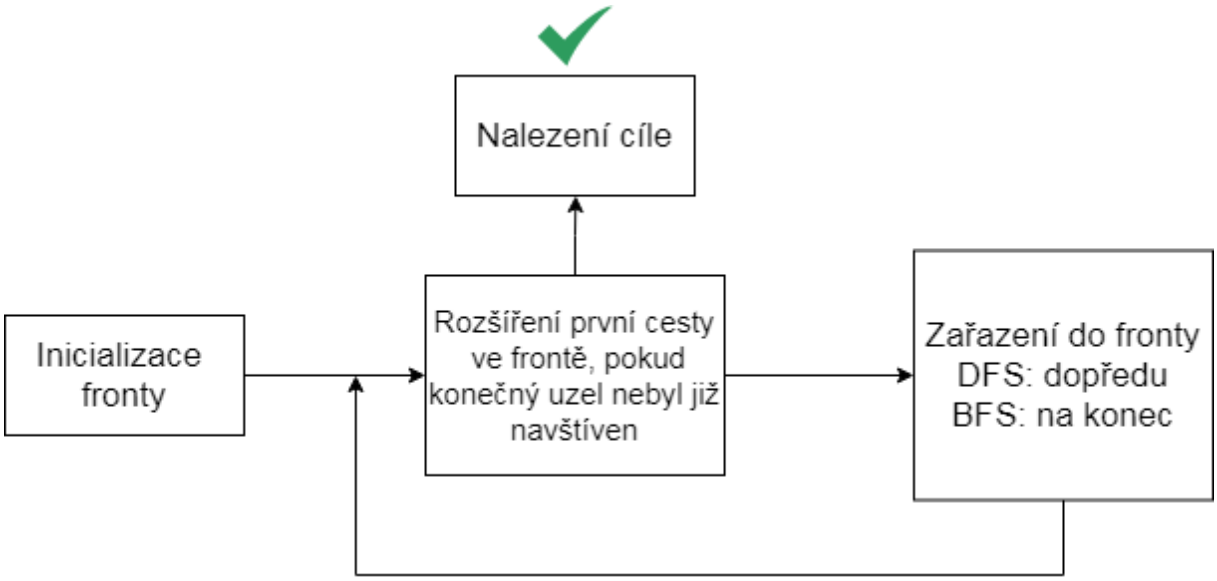
4.5 Efektivnější použití DFS a BFS

Pro efektivnější fungování algoritmů DFS i BFS je zavedena paměť, do které si algoritmus ukládá informace o tom, které uzly již navštívil. Protože nemá smysl expandovat cestu, která vede k stejnému uzlu, který již algoritmus navštívil. Pro porovnání s původním průběhem BFS algoritmu poslouží obrázek 7. Na obrázku lze vidět v druhé úrovni, že algoritmus si pamatuje, že již navštívil vrchol B, proto jej z vrcholu A znovu nerozšiřuje, tedy je přeškrtnut. Obdobně je přeškrtnut bod A, protože byl již jednou navštíven. Je patrné, že díky tomuto opatření je cesta k cíli přímočařejší. [28]



Obrázek 7: Efektivnější průběh BFS algoritmu [28]

V původní diagramu algoritmů BFS a DFS, musela být zavedena podmínka: algoritmus rozšíří první cestu ve frontě pouze tehdy, pokud konečný uzel této cesty nebyl již navštíven. V případě, že podmínka není splněna, algoritmus tuto cestu, již neexpanduje a přeskočí ji. Efektivnější diagram průběhu DFS a BFS je na následujícím obrázku 8.



Obrázek 8: Efektivnější diagram průběhu DFS a BFS algoritmu [28]

5 Rozbor sofistikovanějších algoritmů nejkratší cesty

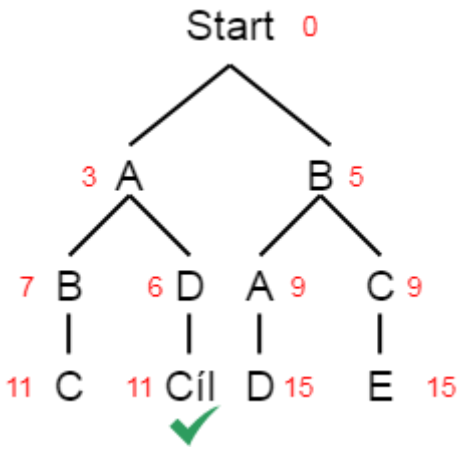
V této kapitole provedu rozbor sofistikovanějších algoritmů, které fungují pouze ve vážených grafech, tedy grafech, kde známe informaci o váze každé hrany (v tomto případě vzdálenosti). Tyto algoritmy si nekladou za úkol pouze najít cestu do cíle, ale snaží se najít nejkratší cestu, tedy cestu s nejnižšími náklady (ve smyslu vzdálenosti).

5.1 Branch and bound algoritmus

Modifikací Branch and bound (B&B) dospějeme ke vzniku algoritmu A*, který je implementován v praktické části.

5.1.1 Ukázka B&B na příkladu

Pro vysvětlení funkce B&B algoritmu použiji opět stejnou ilustraci mapky na Obrázek 2, přičemž zadání úlohy zůstává stejné jako u předešlých algoritmů, tedy začínáme ve vrcholu Start a chceme se dostat do vrcholu Cíl.



Obrázek 9: Diagram průběhu B&B algoritmu [28]

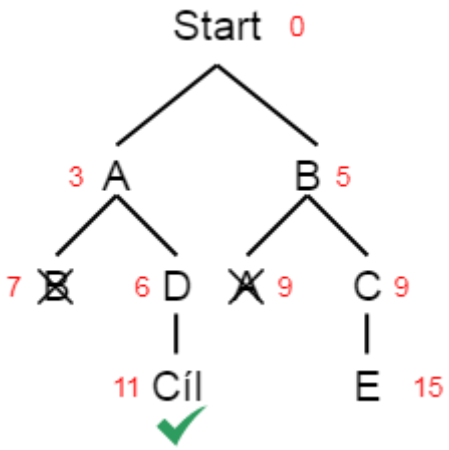
Postup B&B algoritmu popíšu podle obrázku 9. Číselný údaj u vrcholů nám udává akumulovanou vzdálenost, tedy vzdálenost, kterou jsme urazili od vrcholu Start. Akumulovaná vzdálenost v našem případě nemá stanovenou jednotku, obecně ji považujeme za náklady ve smyslu vzdálenosti pro přesun z vrcholu Start do příslušného bodu. Tento algoritmus funguje tak, že expanduje cesty vždy u vrcholu s nejmenší akumulovanou vzdáleností. Z vrcholu Start má dvě možnosti A nebo B. Nyní porovná jejich akumulované vzdálenosti a zjistí, že vrchol A má tuto hodnotu menší než vrchol B, tudíž expanduje cesty bodu A do vrcholů B a D. V další iteraci porovná akumulované vzdálenosti vrcholů B (vzdálenost 5), D a B (vzdálenost 7) a postupuje obdobně dále, až dorazí do vrcholu Cíl. Avšak když dorazí do vrcholu Cíl, musí se přesvědčit, že neexistuje kratší cesta, a proto musí rozvinout stejným způsobem zbytek cest, dokud nebude u ostatních vrcholů akumulovaná vzdálenost větší nebo stejná jako u vrcholu Cíl. [28]

5.2 Vylepšení B&B algoritmu

B&B algoritmus je možné pomocí určitých vylepšení udělat mnohem efektivnějším a rychlejším. V této kapitole vysvětlím dvě techniky vylepšení tohoto algoritmu. Přičemž spojením těchto dvou technik a B&B algoritmu dostaneme nový algoritmus, který se nazývá A* algoritmus (A-star algorithm). [28]

5.2.1 B&B algoritmus a extended list

Pro efektivnější funkčnost B&B algoritmu zavedu tzv. extended list neboli rozšířený seznam, který poslouží jako paměť. Extended list si zapamatovává, které vrcholy už algoritmus expandoval a jakou měli akumulovanou vzdálenost pro případ, že by algoritmus narazil na cestu, která vede do vrcholu, který jíž byl navštíven. V takovém případě porovná akumulované vzdálenosti těchto stejných vrcholů a ten s větší hodnotou přestane dále rozvíjet. Na obrázku 10 můžeme vidět, jak tato technika zjednodušila průběh algoritmu. Algoritmus tedy nemusel zbytečně dále rozvíjet cesty, které by vedly do slepého místa nebo by byly delší než ta optimální. [28]



Obrázek 10: Průběh B&B algoritmu s technikou extended list [28]

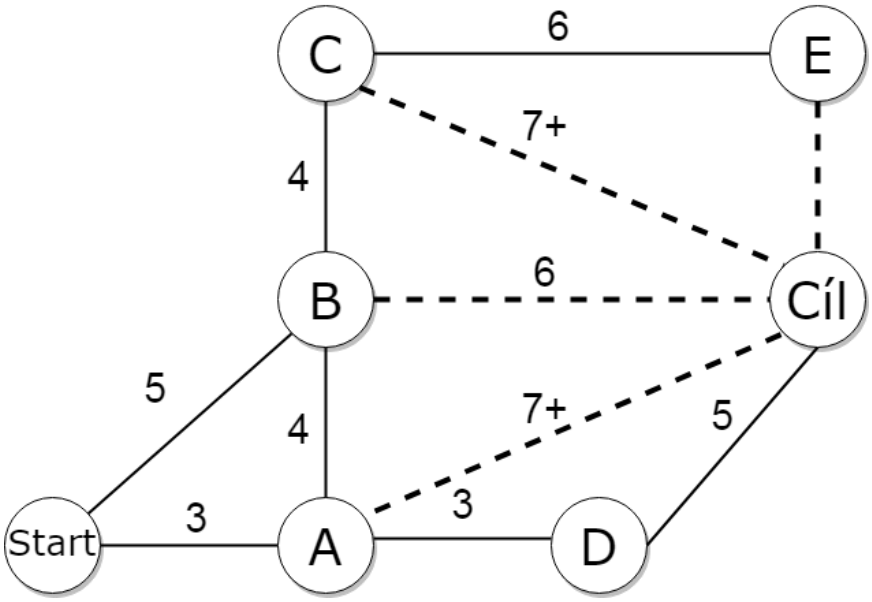
5.2.2 B&B algoritmus a Admissible heuristic

Technika admissible heuristic (přípustná heuristika) kalkuluje s akumulovanou a heuristickou vzdáleností. Když sečteme tyto dvě vzdálenosti, tak dostaneme dolní mez, respektive minimální odhadované náklady (ve smyslu vzdálenosti) na přesun z počátečního bodu do konečného. Díky admissible heuristic víme, že neexistuje kratší cesta jejíž vzdálenost by byla menší než tato stanovená dolní mez. V případě výpočtu z počátečního vrcholu je admissible heuristic rovná pouze heuristické vzdálenosti, protože v počátečním vrcholu je akumulovaná vzdálenost nulová. [28]

Heuristická vzdálenost

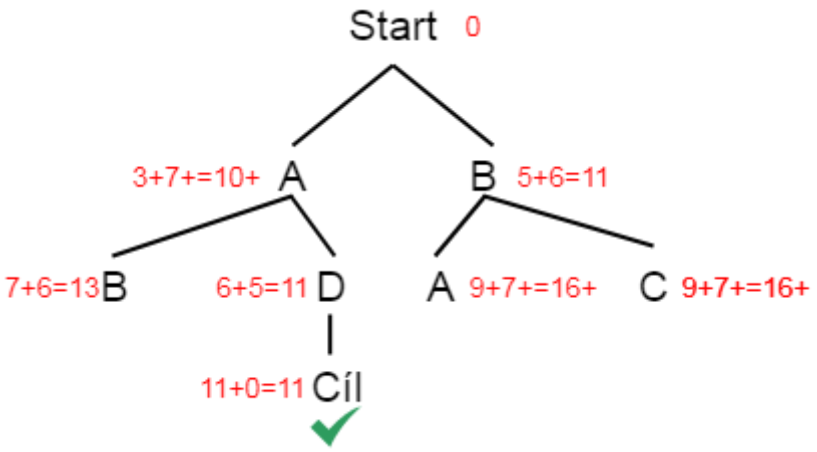
Tato technika je založena na konceptu heuristické vzdálenosti. Jedná se o odhad nejkratší vzdálenosti mezi dvěma místy, která pomyslně směřuje přímým směrem, ačkoliv v některých případech nutně nemusí vést po cestě definované grafem. Existuje více způsobů, jakým se přistupuje k heuristice. Zvolení daného přístupu je vždy závislé na typu dané úlohy a podle ní se také odvíjí výpočet této heuristické vzdálenosti. Na obrázku 11 jsou tyto vzdálenosti vyznačeny přerušovanou čárou. V tomto

konkrétním případě nám zjednodušeně tato vzdálenost říká, jak daleko na pomyslné vzdušné čáře se vrchol nachází od cíle. Je potřeba si uvědomit, že tato heuristická vzdálenost slouží pouze jako odhad, jak daleko přímým směrem se nachází daný vrchol od cíle, protože v některých případech nás může klamat. Například podle obrázku 11 by se mohlo zdát, že bod E je na dobrém místě, protože je postaven nejblíže k vrcholu Cíl, ale v tomto konkrétním případě to není výhodou, protože vrchol E je slepý konec, který nikam nevede. Avšak obecně se dá říct, že je dobré nacházet se blízko cíle. [28]



Obrázek 11: Mapka s heuristickými vzdálenostmi [28]

B&B algoritmus s využitím admissible heuristic probíhá tak, že u každého vrcholu sčítá akumulovanou a heuristickou vzdálenost, respektive určí odhadované minimální náklady (ve smyslu vzdálenosti) na cestu do cíle. Následně rozšiřuje cesty toho vrcholu, který má nejnižší náklady na cestu. Podle diagramu průběhu B&B na obrázku 12 má vrchol A nižší náklady než vrchol B (odhadovaná vzdálenost 11), tudíž expanduje jeho cesty do bodu B (odhadovaná vzdálenost 13) a D. Dále opět musí porovnat náklady vrcholů B (odhadovaná vzdálenost 13), D a B (odhadovaná vzdálenost 11) a postupuje obdobně, dokud nenarazí na vrchol Cíl. V případě, že narazí na 2 vrcholy, které mají stejnou hodnotu nákladu, tak postupuje podle pořadí v abecedě. [28]



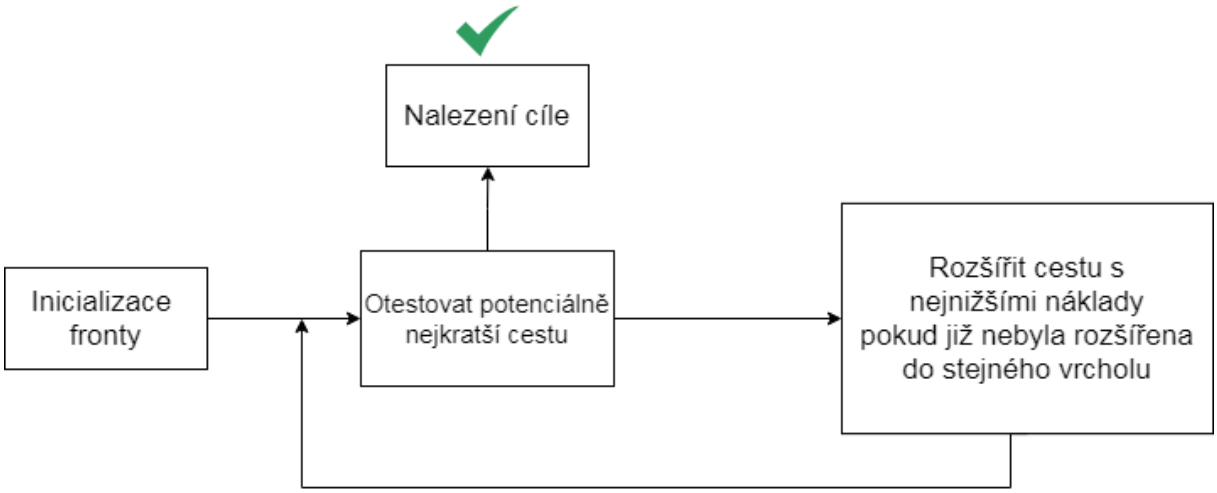
Obrázek 12: Diagram průběhu B&B algoritmu s technikou admissible heuristic [28]

5.3 A* algoritmus

Využitím modifikace Branch and Bound algoritmu spolu s technikami extended list a admissible heuristic dostáváme tzv. A* algoritmus. [28]

5.3.1 Průběh A* algoritmu

Postup A* algoritmu popíšu pomocí vývojového diagramu na obrázku 13. Nejdříve opět proběhne inicializace fronty a poté algoritmus otestuje potenciálně nejkratší cestu, která je zrovna ve frontě. V případě, že by narazil na cíl, tak za předpokladu, že algoritmus vždy testuje potenciálně nejkratší cestu, respektive cestu s nejnižšími náklady (heuristická plus akumulovaná vzdálenost), algoritmus může ukončit hledání, protože ví jistě, že všechny ostatní cesty jsou delší nebo by pouze ve výjimečném případě mohly být stejně dlouhé. Pokud nenarazil zrovna na cíl, tak rozšíří cestu s nejnižšími náklady, za předpokladu, že nesměřuje zrovna do vrcholu, který již byl expandován z jiného místa. Tento proces se opakuje, dokud není nalezen cíl. [28]



Obrázek 13: Diagram průběhu A* algoritmu [28]

6 LabVIEW

Název LabVIEW je zkrácenou formou jeho popisu. Celý název v angličtině zní: Laboratory Virtual Instrument Engineering Workbench. LabVIEW je vizuální programovací jazyk, jehož platforma je určena pro návrh systému. Hlavním cílem tohoto vývojového prostředí bylo umožnit vývoj všech forem systému. [29]

LabVIEW bylo původně vyvinuto společností National Instruments jako pracovní stůl pro řízení testovacích přístrojů. Jeho aplikace se však rozšířily mnohem dál, do celé oblasti návrhu a provozu systému. Dnes se běžně používá pro sběr dat, ovládání přístrojů a průmyslovou automatizaci v různých operačních systémech, včetně Microsoft Windows, různých verzí Unixu, Linuxu a macOS. [29]

LabVIEW je v podstatě prostředí, které umožňuje programování v grafickém jazyce G. Jedná se o grafický programovací jazyk vytvořený společností National Instruments, který byl původně vyvinut pro komunikaci prostřednictvím GPIB, ale od té doby byl značně aktualizován. V dnešní době lze G použít také pro automatizované testovací aplikace a obecný sběr dat programování FPGA.[29]

LabVIEW také poskytuje řadu dalších možností, jako jsou: ladění, automatizovaný multithreading, uživatelské rozhraní aplikace, managment hardwaru a rozhraní pro návrh systému. Tímto způsobem LabVIEW funguje jako portál pro různá zařízení a sdružuje je pod jediný prvek, který se snadno spravuje. [29]

6.1 Popis funkce LabVIEW

Programy, které vytváříme na platformě LabVIEW bývají označovány Virtual Instruments (VI). Většinou bývá program tvořen hlavním VI a v něm samotném se odkazuje neboli se vyvolávají další VI, kterým se přezdívá subVI, to proto, že jsou podřazeny tomu hlavnímu. Pomocí subVI je možno program větvit a dělat ho tak přehlednějším. Jednotlivé VI jsou tvořeny dvojicí obrazovek: čelní panel („Front panel“) a blokový diagram („Block diagram“). Čelní panel představuje podobu uživatelského rozhraní, kde je možné nastavovat a měnit vstupní parametry a stavy pomocí různých ovládacích prvků. Blokový diagram nese logiku celého programu a jsou v něm nadefinovány funkce a operace, které budou implementovány na vstupních parametrech. [29]

6.1.1 Vykonávání kódu v LabVIEW

LabVIEW používá k provádění kódu model data flow neboli tok dat. V aplikaci LabVIEW lze prvek blokového diagramu spustit až po přijetí dat pro všechny požadované vstupy. LabVIEW umožní spuštění každého funkčního bloku, jakmile má k dispozici všechny potřebné vstupy. Když se prvek blokového diagramu provede, může vrátit data, která zase „proudí“ k dalším prvkům blokového diagramu. Tento tok dat se používá k řízenému provádění programu. Právě princip data flow dělá LabVIEW přehlednějším a jednodušším oproti textovým kódům a umožňuje kompilátoru optimalizovat náš systém tak, aby fungoval na paralelním nebo více jádrovém hardwaru. [29]

7 Definice úlohy

Hlavní náplní praktické části této bakalářské práce je implementace algoritmů vyhledávání nejkratší trajektorie v grafickém programovacím jazyku LabVIEW. V praktické části se tedy zaměřuji na implementaci těchto algoritmů: DFS, BFS a A* algoritmus. Principy těchto algoritmů jsem již rozebral v teoretickém rozboru, nyní je aplikuji na konkrétní úlohu. Jedná se o softwarové řešení na teoretické úrovni, proto je nutné na počátku definovat úlohu a stanovit cíle, kterých chci dosáhnout.

Definice úlohy:

Mějme bludiště, ve kterém se někde nachází schovaný poklad v podobě zlaté truhlice. Bludiště je různě rozvětvené a nacházejí se v něm i slepé větve, které nikam nevedou. Do bludiště je vyslán robot, který má za úkol najít nejkratší cestu k pokladu. Součástí úlohy bude možné robota ovládat manuálně, avšak hlavní pozornost bude věnovaná automatickému režimu, kdy robot bude schopný najít cestu k pokladu sám podle zvoleného algoritmu.

Algoritmy DFS a BFS patří do skupiny algoritmů nezávislých na vážených grafech, to znamená, že dokážou hledat cíl nezávisle na znalosti mapy.

Avšak pro implementaci A* algoritmu počítám s tím, že znám mapu a vím, kde se nachází cíl. A* algoritmus je závislý na vážených grafech, respektive v tomto případě vážené mapě, ve které jsou známé vzdálenosti mezi jednotlivými políčky. Vzdálenost neboli náklad na přesun z jednoho políčka do sousedního políčka je 1.

7.1 Reprezentace mapy bludiště

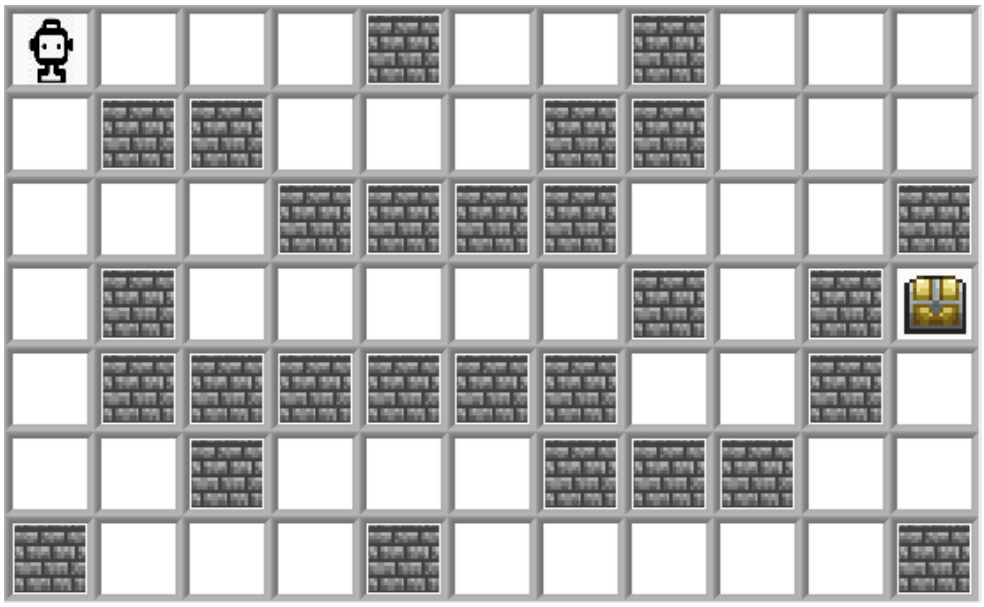
Pro reprezentaci mapy bludiště v LabVIEW jsem využil 2D pole, které si je možné představit jako čtverečkovaný papír neboli 2D grid mapu. V 2D poli je zaveden souřadnicový systém a každé pole má svoje souřadnice a přidělenou hodnotu od 1 do 5, podle toho, co představuje v bludišti. Význam jednotlivých hodnot můžeme vidět v tabulce 2 a jejich vizualizaci na obrázku 14.

Tabulka 2: Význam hodnot v 2D poli

Pořadí:	Význam:
1	Volné políčko
2	Stěna
3	Zvolení startovací pozice robota (pomocné políčko)
4	Poklad (Cíl)
5	Robot

Obrázek 14: Vizualizace hodnot v 2D poli bludiště

Na obrázku 15 je zobrazena vizualizační podoba bludiště, ve kterém se robot pohybuje. Je nutno zmínit, že samotné algoritmy pracují výhradně s číselnou matici, nikoliv obrázkovou, ta slouží pouze pro názornou vizualizaci průběhu, popřípadě výsledného pohybu robota, ke kterému algoritmus dospěje.

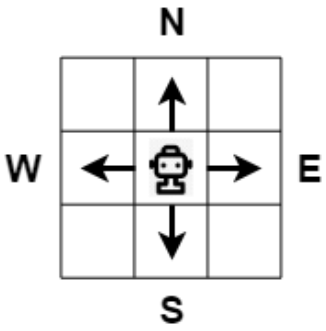


Obrázek 15: Reprezentace bludiště v LabVIEW

V LabVIEW je zavedené indexování od nuly a souřadnice mají tvar (x, y), kde x je číslo sloupce a y je číslo řádku. Indexování políček v 2D poli podle LabVIEW začíná v levém horním rohu 2D pole, tedy podle obrázku 15 se robot nachází na souřadnici (0, 0), přičemž souřadnice x roste v pohybu robota doprava a y souřadnice roste při pohybu dolů.

7.2 Pohyb a ovládání robota v bludišti

Pohybování robota v bludišti je omezeno pouze na 4 směry, podle obrázku 16 se tedy robot může pohybovat pouze nahoru/dolů a doprava/doleva. Jednotlivé směry jsou nazvané podle anglické zkratky odpovídající světové strany (N, E, S, W). Robot se může pohybovat pouze po volných polích prezentovaných hodnotou 1 a může vstoupit na cílové pole s pokladem, které zastupuje hodnota 4.



Obrázek 16: Pohyby robota v bludišti

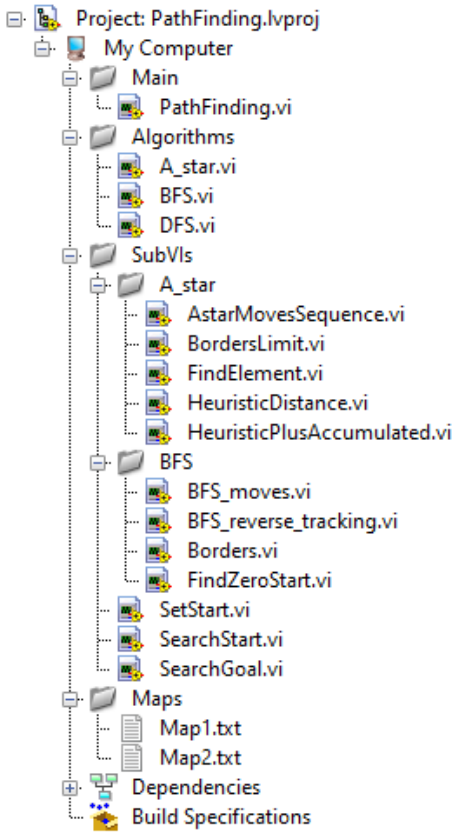
8 Aplikační forma v LabVIEW

Hlavním předmětem praktické části je vytvoření aplikace v LabVIEW pro demonstraci algoritmů pro hledání nejkratší trajektorie, vytvořená aplikace se nachází v příloze A. Definice úlohy a stanovení cílů viz kapitola 7. V této kapitole rozeberu strukturu a funkčnost programu jako celku a později vysvětlím průběhy podprogramů vytvořených pro dané algoritmy, které jsou systematicky rozčleněny do subVI. Následující obrázek 17 znázorňuje členění aplikace do subVI neboli podprogramů.

8.1 Rozbor struktury aplikace

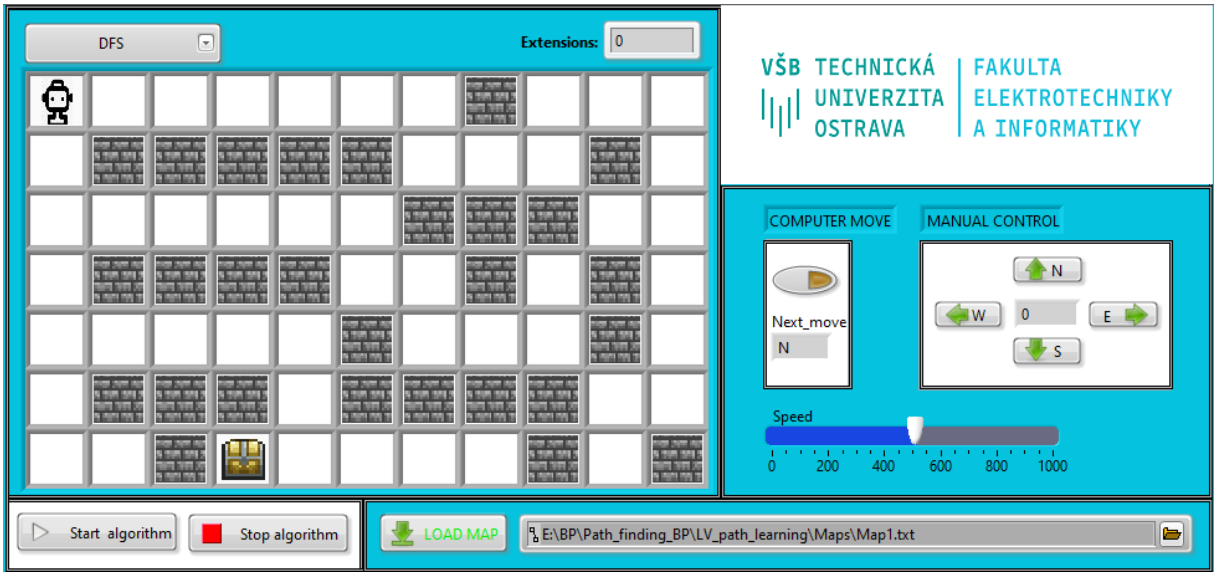
Celková aplikace je zapouzdřena do několika podprogramů subVI. SubVI je samostatné VI, které lze vyvolat jiným VI. SubVI lze chápat jako samostatnou funkci. Používání SubVI je efektivní programovací dovednost, která umožňuje používat stejný kód v různých situacích. V případě takového použití je potřeba správně nastavit tzv. reentrantnost, což je vlastnost, která umožňuje vyvolat jedno subVI současně z několika míst. Velkou výhodou subVI je ušetření místa v blokovém diagramu hlavního programu (Main), díky tomu je program jasný, přehledný a kompaktní. [29]

Z hlavního programu se vyvolávají subVI jednotlivých algoritmů podle aktuálního výběru uživatele, přičemž samotné programy algoritmů jsou rozčleněny do svých vlastních podprogramů subVI, viz následující obrázek 17.



Obrázek 17: Rozčlenění aplikace do subVI

Hlavní program aplikace je PathFinding.vi (součást přílohy A), představuje hlavní uživatelské rozhraní vyvinuté aplikace. Vzhled GUI aplikace je na obrázku 18. Toto rozhraní zprostředkovává uživateli řízení aplikace. Uživatel zde může nastavovat různé parametry, jako například: zvolit mapu bludiště, druh algoritmu a rychlost pro automatický režim, popřípadě ovládat robota manuálně. Součástí je vizualizace pohybu robota po výsledné trase k cíli a počet prověřovaných polí (Extensions), který daný algoritmus vyhodnotí.

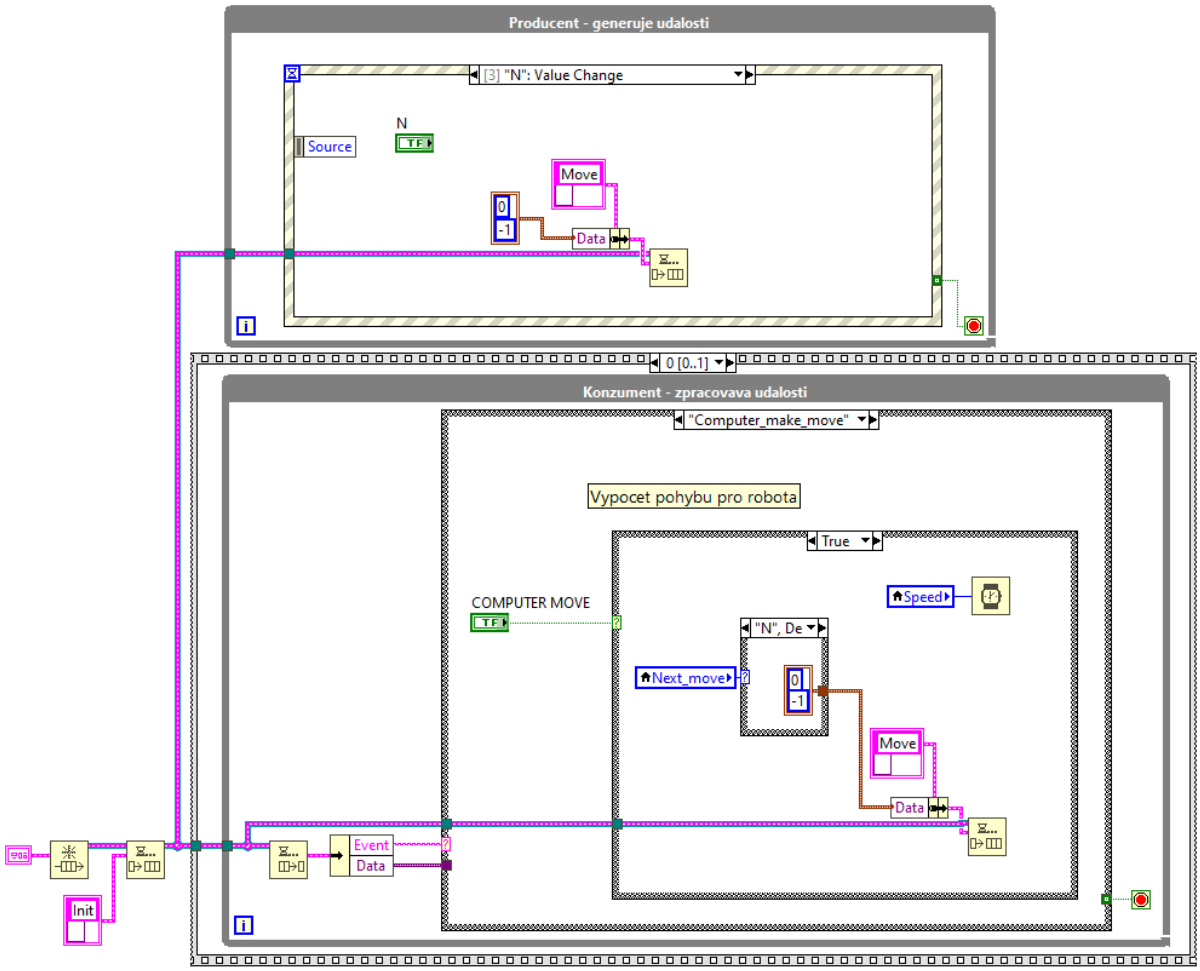


Obrázek 18: Hlavní uživatelské rozhraní aplikace (GUI)

8.2 Forma producent – konzument

Hlavní program vytvořený v LabVIEW je ve formě producent – konzument. Tato struktura program rozděluje do dvou paralelně fungujících smyček while: producent a konzument, viz obrázek 19. Smyčka "producent" obsahuje event strukturu, která okamžitě reaguje na interakce uživatele, jako jsou kliknutí na tlačítka a pohyby myši. Tyto iterace odesílají příkazy pomocí fronty FIFO do smyčky "konzument", která provádí požadované úkoly. Rozdělení stavového automatu na dvě smyčky umožňuje uživatelskému rozhraní setrvat v pohotovosti, pokud úkol vykonávaný spotřebitelem vyžaduje neobvyklou dobu nebo musí počkat, až bude k dispozici sdílený prostředek. [29]

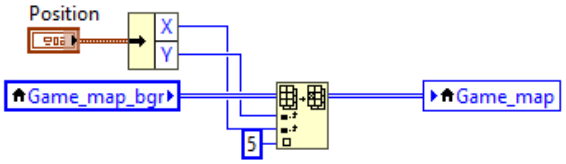
Frontovní systém je založen na bázi FIFO (“First-In, First-Out”). Ve výpočetní technice a v systémové teorii je FIFO (zkratka pro první dovnitř, první ven), slouží jako metoda pro organizaci manipulace s datovou strukturou (často, konkrétně s datovou vyrovnávací pamětí), kde je nejstarší (první) položka zpracována jako první. Takové zpracování je analogické s obsluhou lidí stojících ve frontě na základě zásady „kdo dřív přijde, je dřív na řadě“, tedy lidé jsou obslouženi ve stejném pořadí, v jakém se zařadili do fronty. [30]



Obrázek 19: Forma producent – konzument (kód Mainu)

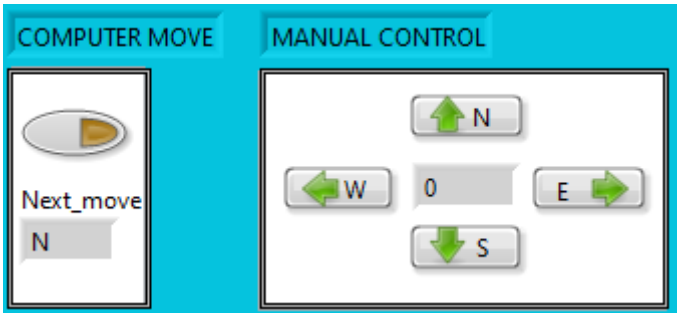
8.2.1 Ovládání robota

Samotné pohybování robota v 2D poli bludiště je v LabVIEW zprostředkováno funkcí Replace Array Subset viz obrázek 20. Prostřednictvím této funkce ve vizualizačním 2D poli bludiště se přesouvá hodnotu 5, tedy hodnota reprezentující robota na příslušné souřadnice daného pole.



Obrázek 20: Kód pro přesouvání robota v bludišti

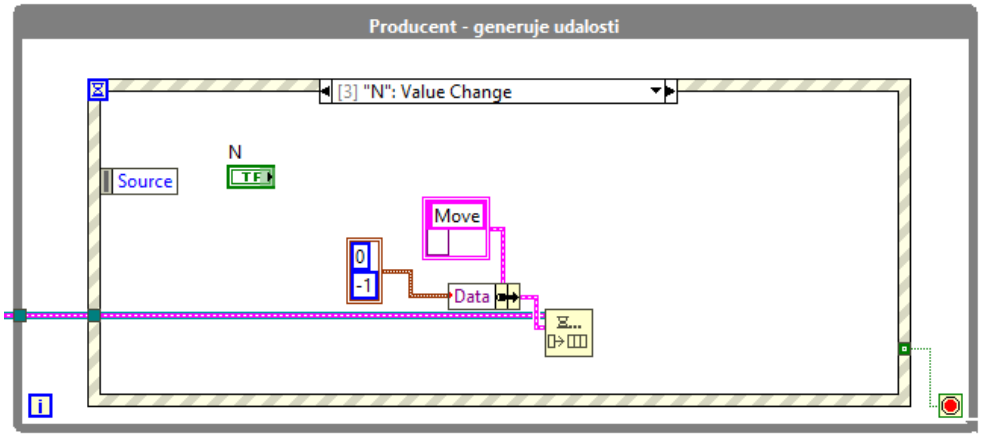
Pro ovládání robota jsem vytvořil dva režimy: manuální a automatický:



Obrázek 21: Ovládání robota GUI

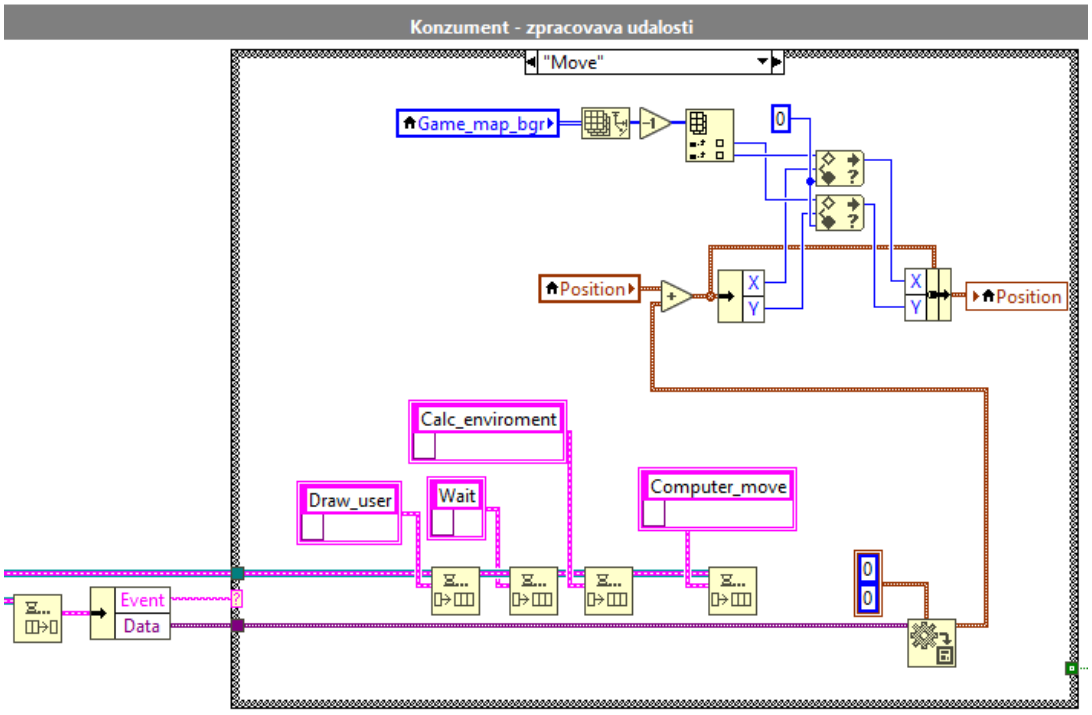
Manuální ovládání:

Manuální ovládání uskutečňuje samotný uživatel aplikace pomocí tlačítek s šipkami, které určují směr pohybu robota, viz obrázek 23 pravá část. Pro jednotlivá tlačítka jsou ve formuli producenta vytvořené příslušné události, ve kterých jsou nadefinovány souřadnice posunu pro jednotlivé směry. Na obrázku 22 je ukázka události pro směr nahoru. Tyto události jsou posílány prostřednictvím fronty FIFO do smyčky konzumenta, kde jsou zpracovávány.



Obrázek 22: Definované události pro jednotlivé směry pohybu robota

Jakmile je událost příslušného pohybu doručena, konzument si ji převezme z fronty FIFO a začne ji zpracovávat podle kódu na obrázku 23. Nejprve souřadnice daného pohybu přičte k souřadnicím, ve kterých se robot aktuálně nachází a ověří, zda se nové, sečtené souřadnice nacházejí v definované oblasti 2D pole, které představuje mapu bludiště. Pokud je tato podmínka splněna, tak přepíše aktuální souřadnice robota na sečtené souřadnice, v opačném případě robot setrvává na svých původních souřadnicích.



Obrázek 23: Zpracování události pohybu v konzumentovi

Automatický režim:

Pro spuštění automatického režimu je potřeba přepnout ovládání na „Computer move“ pomocí stejnojmenného tlačítka a vybrat algoritmus, podle kterého bude program vyhodnocovat pohyb robota. Spuštění algoritmu uskutečníme prostřednictvím tlačítka „Start algorithm“.

Hledání trasy k cíli a průběh vyhodnocování probíhá vždy v příslušném subVI daného algoritmu. SubVI algoritmů posílají zpátky do hlavního programu (Main) pouze jednotlivé pohyby, které má robot vykonat, aby se dostal do cíle a vyhodnocený počet ověřovaných políček (Extensions).

8.3 Rozbor programu jednotlivých algoritmů

V této podkapitole rozeberu a vysvětlím programy jednotlivých algoritmů, které jsou využity v automatickém režimu aplikace. Jejich průběhy vysvětluji pomocí vývojových diagramů.

8.3.1 Průběh programu DFS algoritmu

Pro postup DFS algoritmu hraje klíčovou roli zavedená konvence pořadí směrů pohybů, podle které bude program postupně ověřovat, zda je příslušné políčko daného směru volné, a tedy robot se na něj může přesunout. Zavedená konvence pořadí začíná číslem 0, viz tabulka 3. Dle definice úlohy se robot může pohybovat pouze po volných polích a cílovém poli s truhlou, respektive po polích s hodnotou 1 nebo 4. Smysl ověřování je zde chápan tak, že algoritmus zjišťuje, zda hodnota daného políčka je rovna hodnotě 1 nebo v ideálním případě 4. Postupné ověřování políček probíhá do doby, kdy algoritmus najde první možné políčko, které splňuje tuto podmínku a robot se přesune na nové políčko, kde celý proces ověřování probíhá od nultého směru znova.

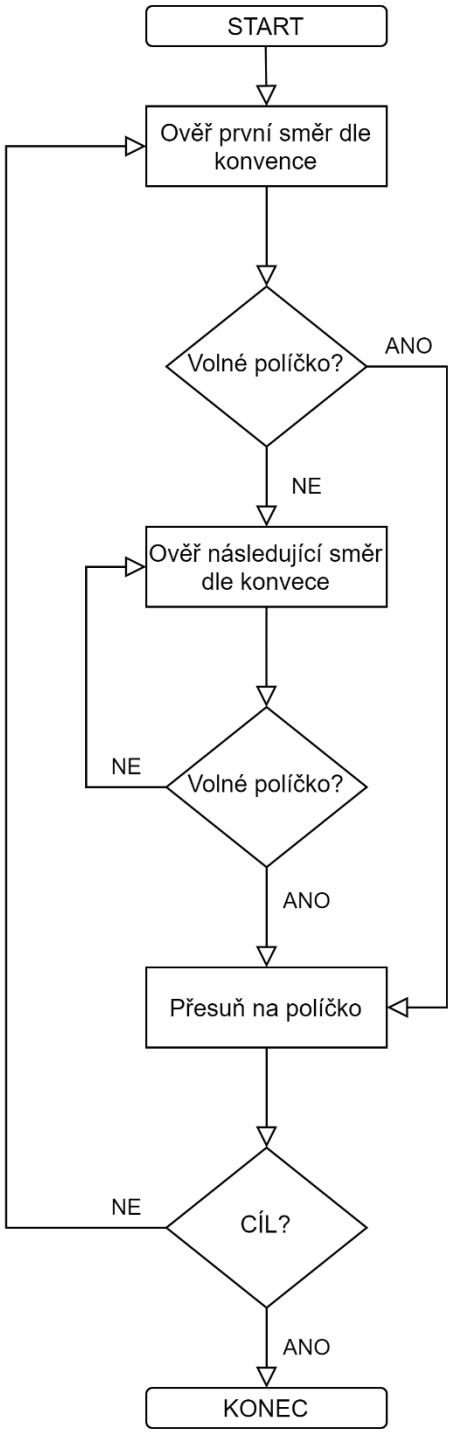
Tabulka 3: Konvence – pořadí směru ověřovaného pole

Pořadí:	Ověřovaný směr:	Podmínka:
0	E – pravé políčko	nenavštíveno
1	S – dolní políčko	nenavštíveno
2	W – levé políčko	nenavštíveno
3	N – horní políčko	nenavštíveno
4	E	
5	S	
6	W	
7	N	

Postup algoritmu popíšu pomocí vývojového diagramu, který je níže na obrázku 24. Jakmile je algoritmus spuštěn, ověří první pohyb v pořadí, tedy konkrétně zjistí, zda je hodnota políčka nacházejícího se napravo od robota rovna 1 nebo 4. Pokud je tato podmínka splněna, tak přesune robota na dané políčko. Jakmile se robot přesune na nové políčko, tak v případě, že se nenachází v cíli, začíná opět ověřovat od začátku stanovené konvence, respektive celý algoritmus se opakuje od začátku. V případě, že je hodnota políčka, na které se přesunul rovna 4, robot dorazil do cíle a algoritmus se ukončí. Pokud však podmínka není splněna a ověřované políčko není volné (není rovno 1), tak algoritmus ověřuje pohyb následující v pořadí a postupuje obdobně, dokud nenarazí na volné políčko, na které se robot může přesunout.

Při ověřování podle konvence algoritmus primárně ověřuje políčka, která nebyla ještě navštívena. Pokud se v jeho okolí již nenachází nenavštívené políčko, tak postupuje v konvenci pořadí a hledá první možné políčko na které se může přesunout, ačkoliv už navštíveno bylo. A takto postupuje, dokud se nevrátí do místa, kde měl na výběr více možností, které doposud nenavštívil. Tato metoda se nazývá backtracking, tedy zpětné vyhledávání.

Algoritmus postupuje vždy stejně, ačkoliv se robot nachází na kterémkoliv volném políčku bludiště. Algoritmus se zastaví ve chvíli, kdy robot dorazí do cíle. Průběh DFS algoritmu mapou 1 je zaznamenán v podobě videa, viz příloha B1.



Obrázek 24: Vývojový diagram programu DFS algoritmu

8.3.2 Průběh programu BFS algoritmu

Průběh BFS algoritmu je rozdělen do tří procesů, které na sebe navazují a vykonávají se postupně po jednom. Každý z těchto procesů má vytvořené samostatné subVI. Procesy a jejich pořadí jsou uvedeny níže. Význam funkci jednotlivých procesů vysvětlím pomocí vývojových diagramů.

1. BFS – vyhledávání cíle
2. BFS reverse tracking – zpětné vyhledávání cesty
3. BFS moves – sekvence pohybů

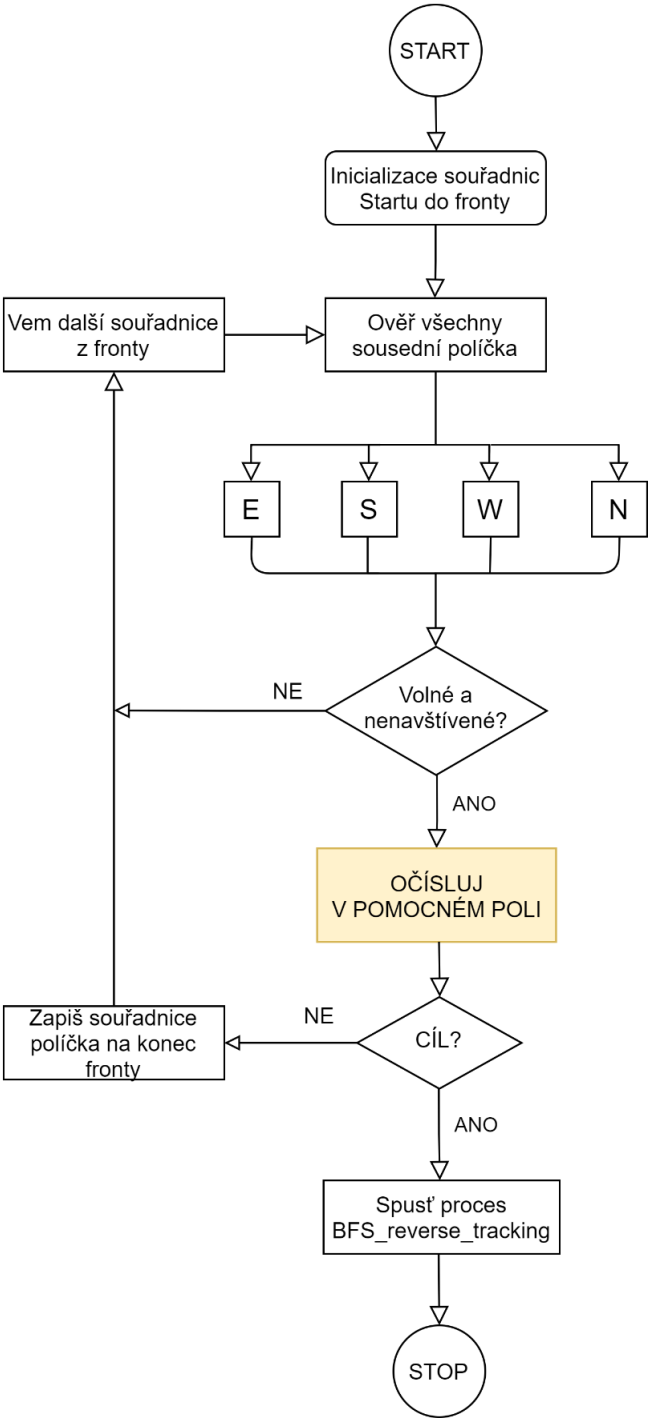
První proces má za úkol najít cíl v bludišti a jeho souřadnice. Postup, jakým řeší tento úkol vysvětlím podle vývojového diagramu níže na Obrázek 26. Jakmile je algoritmus spuštěn, proběhne inicializace fronty, přičemž je do ní vložena první položka, a to sice souřadnice políčka, ze kterého robot startuje. V LabVIEW tuto frontu reprezentuje 1D pole. Jelikož souřadnice políčka v bludišti je určena dvěma čísly, každá položka souřadnice má svoji samostatnou frontu, tedy do jedné fronty se ukládá položka x, do druhé y. Následně algoritmus ověří všechny 4 sousední políčka od toho startovního, na které by se robot mohl potenciálně přesunout. Postup ověřování je pro všechny směry (E, S, W, N) stejný, viz diagram na Obrázek 26. Nejdříve zjišťuje, zda se jedná o nenavštívené políčko a zda je dané políčko volné, tedy zda je jeho hodnota 1 nebo 4 v případě, že se jedná o cíl. Pokud políčko splňuje podmínku a je volné, očísluje si ho v pomocném poli, princip číslování vysvětlím následně. Toto číslování později hraje klíčovou roli v postupu druhého procesu, právě proto je tato operace zvýrazněna ve vývojovém diagramu. Pokud políčko bylo již ověřováno nebo není volné, tak jej algoritmus přeskočí. V případě, že ověřované políčko je cílové, tak se spustí druhý proces a ten první se zastaví. V opačném případě, tedy když dané políčko není cílové, tak zapíše souřadnice do fronty, vezme následující položku souřadnic z fronty a celý postup se opět opakuje.

Princip číslování

Jakmile je BFS algoritmus spuštěn, a tedy začne proces 1, tak je vytvořeno pomocné pole, které má stejnou velikost, jako mapa bludiště a každé políčko má hodnotu nekonečna, to je z důvodu usnadnění implementace procesu 2. V pomocném poli je do políčka odpovídajícímu pozici startovního políčka bludiště vložena hodnota 0. Samotné číslování funguje tak, že políčku, které má být očíslováno je udělena hodnota o 1 vyšší než hodnota políčka, ze kterého bylo ověřeno, viz obrázek 25. Toto číslování zároveň určuje úroveň BFS algoritmu, ve které bylo dané políčko ověřeno.

0	1	2	3	Inf	7	8	Inf	Inf	Inf	Inf
1	Inf	Inf	4	5	6	Inf	Inf	Inf	Inf	Inf
2	3	4	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
3	Inf	5	6	7	8	9	Inf	Inf	Inf	Inf
4	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	20
5	6	Inf	10	11	12	Inf	Inf	Inf	18	19
Inf	7	8	9	Inf	13	14	15	16	17	Inf

Obrázek 25: Číslování políček v pomocném poli



Obrázek 26: Vývojový diagram prvního procesu BFS

2. proces

Druhý proces se spustí ve chvíli, kdy je 1. proces dokončen. Význam tohoto procesu je vyhledání cesty, která vede ze startovního políčka do cíle neboli vytvoření posloupnosti souřadnic jednotlivých políček, které musí robot navštívit, aby došel do cíle. Vyhledávání cesty v tomto procesu je výhradně založeno na operování s pomocným polem, které bylo vytvořeno a očíslováno v procesu 1.

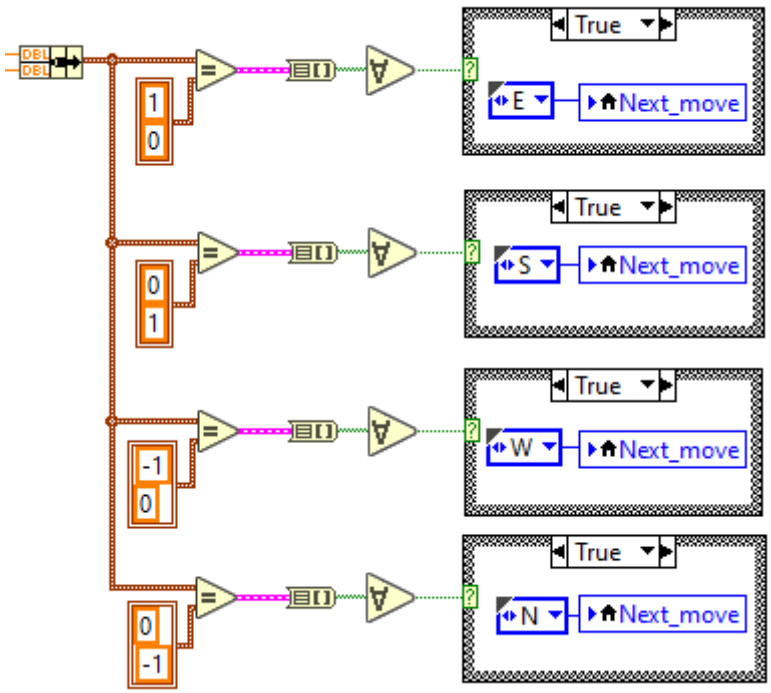
Postup procesu 2 vysvětlím podle vývojového diagramu, který se nachází níže na Obrázek 28. Jakmile se proces 2 spustí, tak je opět inicializovaná nová fronta, do které jsou tentokrát vloženy souřadnice cílového políčka, které byly zjištěny v prvním procesu. Následně porovná všechny 4 sousední políčka tohoto cílového políčka a vybere to s nejmenší hodnotou. Políčko s nejmenší hodnotou zapíše do fronty a vezme následující souřadnice z fronty, tedy ty, které do ní zapsal a celý postup porovnávání probíhá znovu. Toto se opakuje do té doby, dokud nenarazí na startovní políčko, tedy políčko s hodnotou 0 a následně spustí třetí proces.

3. proces

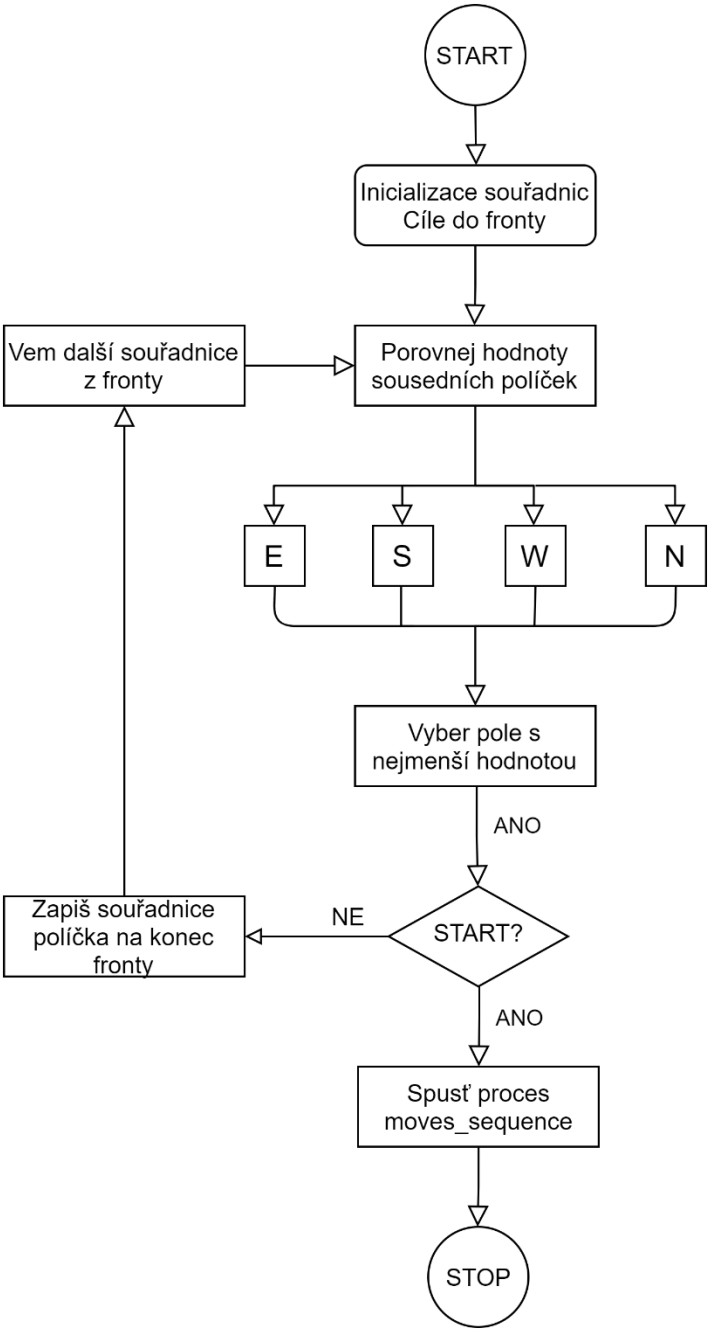
V tomto procesu algoritmus ze sekvence souřadnic políček, které vedou do cíle, vyhodnocuje předdefinované pohyby (E, S, W, N) a posílá je do hlavního programu, kde jsou uskutečňovány. Celý program BFS algoritmu kromě procesu 3 proběhne na pozadí a až vyhodnotí správnou cestu do cíle, tak souběžně s procesem 3 v hlavním programu proběhne výsledný pohyb robota do cíle. Oproti tomu program DFS algoritmu promítá své postupovaní do hlavního programu po celou dobu, co probíhá.

V LabVIEW je tento proces realizován tak, že fronta souřadnic políček vedoucích z cíle do startu se celá překlopí, aby byla sekvence souřadnic seřazena od startovního políčka. Následné vyhodnocování pohybů probíhá tak, že vždy od vyhodnocované souřadnice se odečte souřadnice předešlá a následně podle logiky programu na obrázku 27 je určen daný pohyb (E, S, W, N), který má robot vykonat.

Ukázka, jak postupuje program BFS algoritmu v případě mapy 1 je zaznamenána na videu, viz příloha B2. Vizualizace tohoto průběhu proběhla v samotném subVI BFS.vi.



Obrázek 27: Vyhodnocovací logika pro určení pohybu



Obrázek 28: Vývojový diagram procesu BFS reverse tracking

8.3.3 Průběh A* algoritmus

Průběh A* algoritmu je rozdělen do dvou procesů, které na sebe navazují a vykonávají se postupně po jednom. Každý z těchto procesů má vytvořené samostatné subVI. Procesy a jejich pořadí jsou uvedeny níže. Názvy těchto procesů odpovídají názvu jejich subVI.

1. A_star – vyhledávání nejkratší cesty k cíli
2. AstarMovesSequence – vyhodnocení sekvence pohybu

Snaha A* algoritmu je vyhledávat nejkratší cestu k cíli, pokud možno co nejefektivněji, a tedy zbytečně neprozkoumávat políčka navíc. K tomu využívá techniku admissible heuristic, viz kapitola 5.2.2. Obecně existuje více způsobů, jak počítat heuristickou vzdálenost a jejich použití je závislé na definici pohybové logiky úlohy. V definici mé úlohy je stanoveno, že se robot může pohybovat pouze čtyřmi směry, viz 7.2, z toho důvodu v programu využívám způsob výpočtu heuristických vzdáleností, známý jako Manhattan distance, vysvětlím níže.

Admissible heuristic (minimální odhadované náklady)

Neboli minimální náklady f(n) na odhadovanou nejkratší cestu z políčka start do cíle se vypočtou jako součet heuristické vzdálenosti h(n) a akumulované vzdálenosti g(n), viz rovnice níže, kde n určuje políčko, pro které jsou dané vzdálenosti počítány. [28]

$$f(n) = h(n) + g(n) \quad (1)$$

[28]

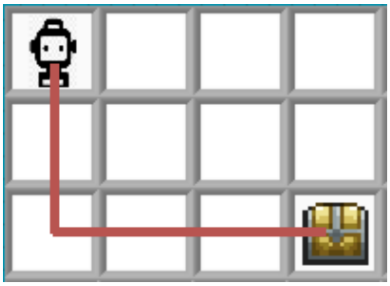
Manhattan heuristic distance

Manhattanská dálková heuristika h(n) se počítá jako součet absolutních hodnot rozdílů v souřadnicích x a y cíle a souřadnic x_1 a y_1 aktuálního políčka, viz rovnice 2, kde n je aktuální políčko. [27]

$$h(n) = |x - x_1| + |y - y_1| \quad (2)$$

[27]

Manhattanská dálková heuristika je znázorněna na následujícím obrázku červenou křivkou. Tato heuristika udává nejnižší odhadovanou vzdálenost do cíle z aktuálního políčka, přičemž bere v potaz stanovenou pohybovou logiku úlohy. Tento výpočet probíhá v samostatném subVI HeuristicDistance, součást přílohy A.



Obrázek 29: Manhattanská dálková heuristika

Akumulovaná vzdálenost

Pro výpočet akumulované vzdálenosti g(n) využívám rovněž způsob výpočtu Manhattan distance, který jsem pouze trochu pozměnil. Ve výpočtu jsem pouze nahradil souřadnice cíle x a y souřadnicemi startovního políčka x_0 a y_0, viz rovnice 3 níže. [27]

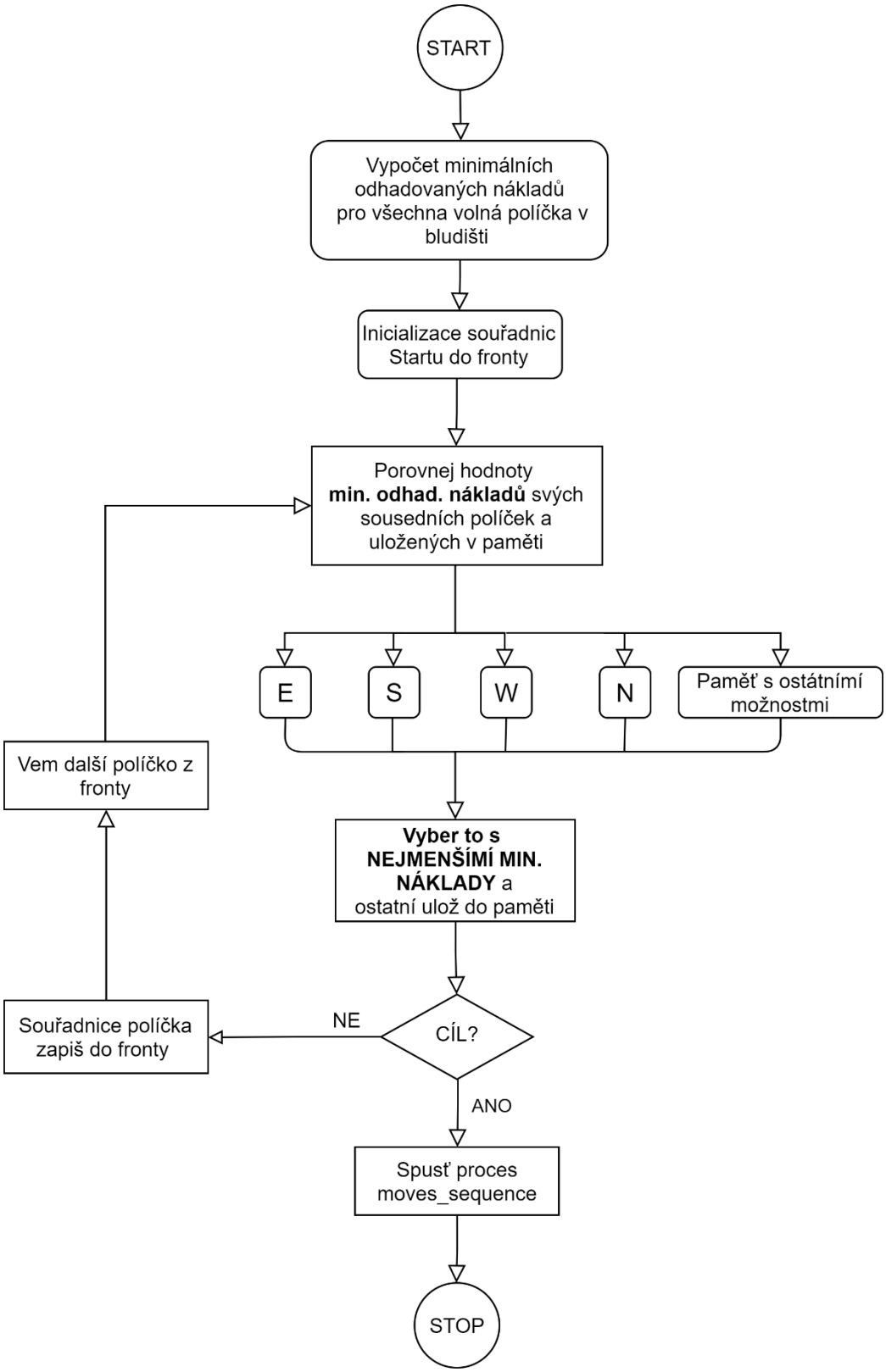
$$g(n) = |x - x_0| + |y - y_0| \quad (3)$$

[27]

Podle vývojového diagramu programu A* algoritmu na Obrázek 30 popíšu, jak program postupuje. První proces má za úkol najít nejkratší cestu do cíle, a přitom neprověřovat zbytečná políčka navíc. Ve chvíli, kdy je algoritmus spuštěn, tak nejdříve proběhne výpočet minimálních odhadovaných nákladů podle rovnice 1 pro každé políčko v bludišti, které zrovna nereprezentuje zeď. Následně proběhne inicializace startovních souřadnic do fronty a začne proces porovnávání minimálních odhadovaných nákladů sousedních políček a políček uložených v paměti. Algoritmus vybírá vždy políčko s nejmenší hodnotou minimálních odhadovaných nákladů a ostatní sousední políčka a jejich hodnoty uloží do paměti. Následně ověří, zda vybrané políčko je cílové, pokud ano, tak spustí proces 2. V opačném případě zapíše souřadnice daného políčka do fronty. Poté si vytáhne následující políčko z fronty a proces porovnávání hodnot minimálních nákladu sousedních a uložených políček se opakuje. Toto se celé opakuje do doby, než algoritmus objeví cíl a spustí 2. proces.

V případě, že nejmenší porovnávaná hodnota je u více ověřovaných políček stejná, tak algoritmus jedno vybere zpravidla dle zavedené konvence pořadí směrů (E, S, W, N), tedy prioritu rozvíjení má směr E. Cestu pokračuje rozvíjet ve směru vybraného políčka do chvíle, dokud některé políčko uložené v paměti nemá menší hodnotu minimálních odhadovaných nákladů než to zrovna uvažované. V případě, že pří procesu porovnávání má některé políčko v paměti menší hodnotu, tak algoritmus na něj přesedlá.

Druhý proces pouze ze sekvence souřadnic políček, které vedou do cíle, vyhodnocuje předdefinované pohyby (E, S, W, N) a posílá je postupně do hlavního programu, kde jsou uskutečňovány. Tento proces je obdobný jako BFS moves – sekvence pohybů, který jsem již podrobněji vysvětlil, viz 3. proces. V tomto procesu je oproti 3. procesu BFS jen mála změna, sekvence souřadnic se nemusí překlápět, protože je řazena od startovního pole. Jakmile robot dorazí do cíle, algoritmus se ukončí. Výsledek programu A* algoritmu pro mapu 1 je zaznamenán na videu, viz příloha B3.



Obrázek 30: Vývojový diagram průběhu programu A* algoritmu

9 Zhodnocení výsledků aplikovaných algoritmů

Za účelem ověření správné funkčnosti, variabilnosti a porovnání efektivnosti vytvořených programů algoritmů jsem tyto programy testoval na třech různých mapách bludiště. Jednotlivé mapy bludiště jsou zobrazeny níže na Obrázek 31. Všechny tři mapy jsou v podobě čtvercové mřížky o rozměrech 7 x 11, tedy celkem mají vždy 77 políček. Menší velikost mapy byla zvolena záměrně, aby bylo možné jednoduše pouhým okem ověřit výsledek algoritmu. Mapy obsahují slepá místa a rozcestí, která by mohla algoritmy mást. Právě na těchto místech a rozcestích je možné pozorovat, jak se daný algoritmus rozhoduje a postupuje. Výsledný průběh jednotlivých algoritmů pro mapu 1 je zaznamenán ve formě videa, viz příloha B.

Kritéria, podle kterých porovnávám jejich efektivnost jsou počet prověřených polí nutný k objevení cíle a zda je nalezená cesta k cíli ta nejkratší. První zmíněné kritérium se v aplikaci automaticky počítá v průběhu programu podle počtu navštívených (prověřených) políček a je uvedeno v horní části GUI aplikace vedle názvu Extensions, viz Obrázek 18. Druhé kritérium bylo testováno na mapě 3, výsledek je níže rozebrán.

Jako jiné kritérium se nabízí doba trvání algoritmu, ale to by mohlo být zavádějící z důvodu použití různých způsobů implementace, proto jsem toto kritérium nebral v úvahu.

V tabulce 4 jsou zjištěné výsledky prvního stanoveného kritéria, které bylo sledováno na 3 zmíněných mapách pro všechny 3 aplikované algoritmy.

Tabulka 4: Výsledky hodnoceného kritéria testovaných map

Algoritmus	Počet prověřených polí (Extensions)		
	Mapa 1	Mapa 2	Mapa 3
A*	19	36	13
BFS	34	36	43
DFS	45	36	15

Ze zaznamenaných výsledků v tabulce 4 vyplývá následující:

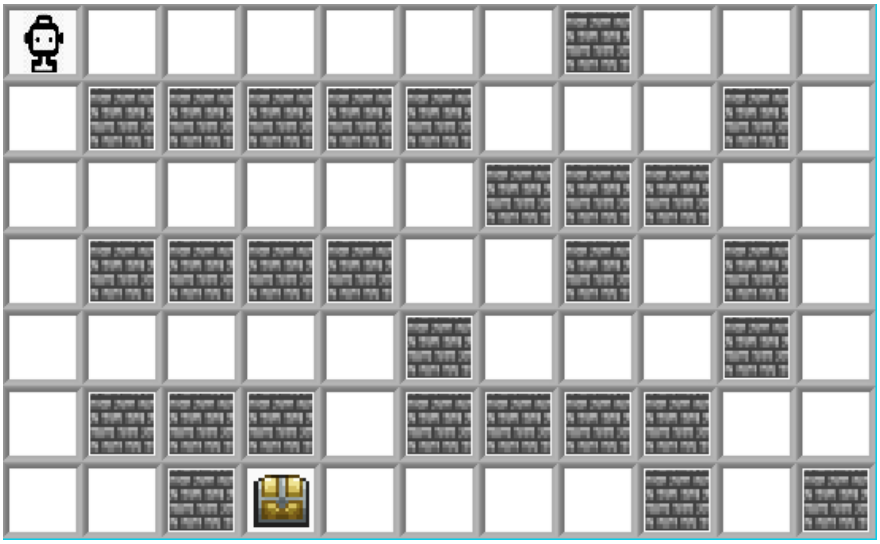
V celkovém vyhodnocení výsledků v rámci těchto tří konkrétních testovaných map nejefektivněji postupuje A* algoritmus, protože k nalezení cíle potřeboval ověřit nejméně políček, a to sice ve dvou případech: mapy 1 a mapy 3. Tento závěr jsem přepokládal na začátku vytváření aplikace a zde se potvrdil zjištěnými výsledky.

Mapa 2 demonstruje případ, ve kterém všechny tři algoritmy musely ověřit stejný počet políček, aby nalezly cíl, tedy jinými slovy byly stejně efektivní.

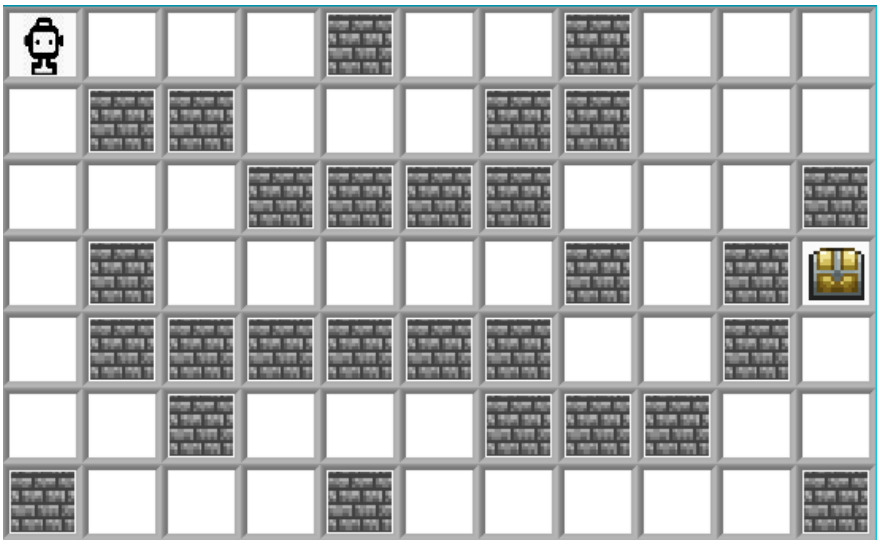
Mapa 3 byla zaměřená na sledování, jak se algoritmy zachovají, když mají na výběr více cest, které vedou do cíle a zda se jim podaří zvolit nejkratší cestu. Algoritmům A* a BFS se podařilo zvolil tu nejkratší cestu. DFS algoritmus zvolil až 2. nejkratší cestu (proto je výsledek v tabulce 4 zvýrazněn červenou barvou), avšak pro nalezení cíle potřeboval prověřit méně políček než BFS. Algoritmy BFS a DFS nejsou vždy vhodné pro vyhledávání nejkratší trajektorie, zároveň nelze říct, že vždy určí právě tu nejkratší cestu. Jejich hlavní využití je prozkoumávání grafů, sítí a podobných prostředí, avšak z jejich

fundamentálního základu vycházejí sofistikované algoritmy, jako je A* algoritmus, které jsou určené právě k vyhledávání nejkratší trajektorie.

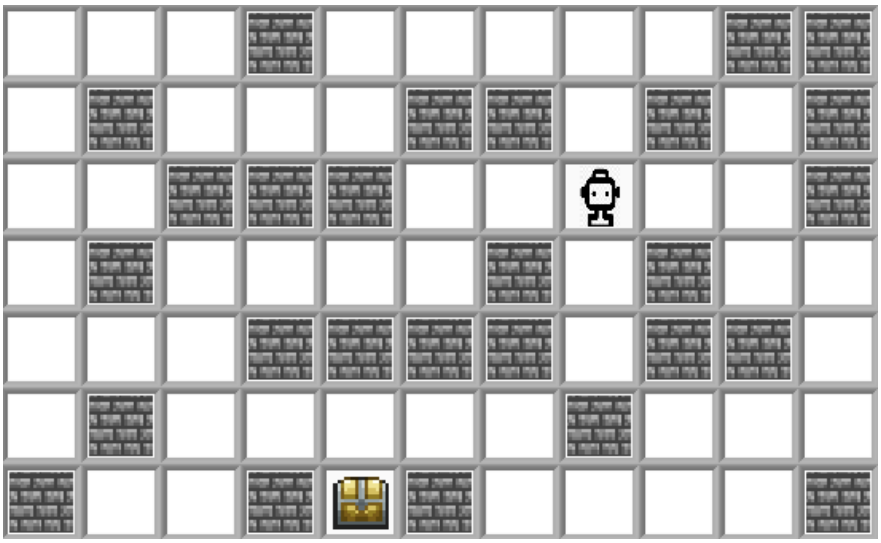
Co se týče porovnání BFS a DFS, tak nemůžu jednoznačně říct, který z nich je efektivnější. Každý z nich má situaci, ve které by byl efektivnější než ten druhý. Ačkoliv v případě mapy 3 byl z pohledu počtu prověřených polí efektivnější DFS, tak zvolil delší cestu, a ne tu nejkratší. V případě, že je mým požadavkem nalezení nejkratší cesty do cíle, nehledě na to, že hledání bude trvat delší dobu, respektive bude nutné prověřit více polí, tak můžu říct, že v rámci hodnocení na těchto třech konkrétních mapách je efektivnější BFS než DFS.



Obrázek 31: Mapa bludiště 1



Obrázek 32: Mapa bludiště 2



Obrázek 33: Mapa bludiště 3

10 Závěr

Cílem mé bakalářské práce bylo seznámení se problematikou určení optimální trajektorie pomocí metod umělé inteligence, a následné zpracování algoritmů, v grafickém programovacím prostředí LabVIEW. K úspěšnému řešení této problematiky existuje široká škála algoritmů. V této práci jsem představil ty nejznámější a nejpoužívanější algoritmy, avšak podrobněji jsem se věnoval pouze těmto algoritmům: Depth-first search, Breadth-first search a A* algoritmu. První dva zmíněné patří k fundamentálním algoritmům a jsou základním stavebním kamenem pro ty pokročilejší a efektivnější algoritmy, jako je například A* algoritmus, který je v mnoha případech stále považován za nejlepší řešení a má v současnosti širokou oblast využití.

V praktické části jsem zmíněné algoritmy na základě teoretických podkladů aplikoval a modifikoval pro konkrétní definovanou úlohu. Implementace algoritmů a vytvoření aplikace proběhlo v grafickém programovacím jazyce LabVIEW. Výstupem mé práce je funkční aplikace, která vizuálně simuluje úlohu robota v bludišti, jehož úkolem je najít cestu k cíli. Robota lze ovládat jak automaticky podle zmíněných algoritmů, tak manuálně prostřednictvím tlačítek šipek. Součástí aplikace jsou 3 vytvořené mapy, na který je funkčnost aplikovaných algoritmů ověřená. Tyto 3 mapy lze snadno do 2D pole nahrát pomocí tlačítka „Load map“. Uživatel aplikace má také možnost vytvořit si svou vlastní mapu, na které lze pozorovat a dále testovat průběhy implementovaných algoritmů.

Na základě testování algoritmů na 3 různých mapách jsem ověřil funkčnost a variabilnost vytvořených programů pro dané algoritmy. Testování potvrdilo, že všechny algoritmy fungují správně a splnily predikované výsledky, které jsem očekával při vytváření jejich programů. Součástí testování bylo porovnání efektivity jednotlivých algoritmů, které bylo podloženo zjištěnými výsledky v Tabulka 4. Zhodnocení výsledků bylo podrobněji rozebráno v Zhodnocení výsledků aplikovaných algoritmů. Na základě zhodnocení výsledků jsem dospěl k těmto závěrům: Efektivita jednotlivých algoritmů závisí na typu mapy. Nejlepší možný výsledek je shodný, nejhorší se dimenzionálně liší. Existují různé mapy, na kterých je použití daného algoritmu různě efektivní. Také existují typy map, na kterých jsou všechny 3 algoritmy stejně efektivní, viz jako příklad mapa 2 na Obrázek 32.

Výsledná aplikace je rozšířitelná, škálovatelná, doplnitelná a splňuje standardy NI. V současné podobě se jedná o systém výhradně softwarového charakteru a aplikace může posloužit na akademické půdě jako demonstrace průběhu algoritmů nejkratší trajektorie.

Co se týče vize dalšího vývoje, jakým by se práce mohla dále vyvíjet, tak při využití patřičných doplnění a modifikací by práce mohla posloužit jako inspirace pro praktickou realizaci systému určeného pro autonomní řízení robota v reálném prostředí.

Literatura

[1] Angles, R., & Gutierrez, C. (2008). Survey of graph database models. ACM Computing Surveys, 40(1), 1-39. Dostupné z: https://doi.org/10.1145/1322432.1322433

[2] Zhang, W., Zhang, R., Shang, R., Li, J., & Jiao, L. (2019). Application of natural computation inspired method in community detection. Physica A: Statistical Mechanics and its Applications, 515, 130-150. Dostupné z: https://doi.org/10.1016/j.physa.2018.09.186

[3] Gao, J., & Gao, J. (2019). A Similarity Measurement Method Based on Graph Kernel for Disconnected Graphs. In Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (pp. 6430-6431). International Joint Conferences on Artificial Intelligence Organization. Dostupné z: https://doi.org/10.24963/ijcai.2019/904

[4] Hamilton W.L., Ying R., Leskovec J.Representation learning on graphs: Methods and applications (2017) Dostupné z: https://arxiv.org/abs/1709.05584

[5] Ma, L., Gong, M., Liu, J., Cai, Q., & Jiao, L. (2014). Multi-level learning based memetic algorithm for community detection. Applied Soft Computing, 19, 121-133. Dostupné z: https://doi.org/10.1016/j.asoc.2014.02.003

[6] Shang, R., Liu, H., Jiao, L., & Esfahani, A. M. G. (2017). Community mining using three closely joint techniques based on community mutual membership and refinement strategy. *Applied Soft Computing*, *61*, 1060-1073. https://doi.org/10.1016/j.asoc.2017.08.050

[7] Gong, M., Ma, L., Zhang, Q., & Jiao, L. (2012). Community detection in networks by using multiobjective evolutionary algorithm with decomposition. *Physica A: Statistical Mechanics and its Applications*, *391*(15), 4050-4060. https://doi.org/10.1016/j.physa.2012.03.021

[8] Newman, M. E. J. (2006). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, *103*(23), 8577-8582. https://doi.org/10.1073/pnas.0601602103

[9] Roweis, S. T. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, *290*(5500), 2323-2326. https://doi.org/10.1126/science.290.5500.2323

[10] Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. (2002). In T. G. Dietterich, S. Becker, & Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems 14*. The MIT Press. https://doi.org/10.7551/mitpress/1120.003.0080

[11] D. Luo, F. Nie, H. Huang, C.H. Ding, Cauchy graph embedding, in: Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, pp. 553-560. Dostupné z: https://openreview.net/forum?id=r1VmZ3-OZH

[12] Ou, M., Cui, P., Pei, J., Zhang, Z., & Zhu, W. (2016). Asymmetric Transitivity Preserving Graph Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1105-1114). ACM. https://doi.org/10.1145/2939672.2939751

[13] Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). DeepWalk. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 701-710). ACM. Dostupné z:https://doi.org/10.1145/2623330.2623732

[14] Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). DeepWalk. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 701-710). ACM. Dostupné z: https://doi.org/10.1145/2623330.2623732

[15] H. Chen, B. Perozzi, Y. Hu, S. Skiena, Harp: Hierarchical representation learning for networks, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018. Dostupné z: https://www.researchgate.net/publication/317930440_HARP_Hierarchical_Representation_Learning_for_Networks

[16] Derhami, V., Khodadadian, E., Ghasemzadeh, M., & Zareh Bidoki, A. M. (2013). Applying reinforcement learning for web pages ranking algorithms. *Applied Soft Computing*, *13*(4), 1686-1692. Dostupné z: https://doi.org/10.1016/j.asoc.2012.12.023

[17] S. Cao, W. Lu, Q. Xu, Deep neural networks for learning graph representations, in: Thirtieth AAAI Conference on Artificial Intelligence, 2016. Dostupné z: https://arxiv.org/pdf/2001.00293.pdf

[18] Kipf T.N., Welling M.Semi-supervised classification with graph convolutional networks (2016) Dostupné z: https://openreview.net/pdf?id=SJU4ayYgl

[19] Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., & Mei, Q. (2015). LINE. In *Proceedings of the 24th International Conference on World Wide Web* (pp. 1067-1077). International World Wide Web Conferences Steering Committee. https://doi.org/10.1145/2736277.2741093

[20] Hougardy, S. (2010). The Floyd–Warshall algorithm on graphs with negative cycles. *Information Processing Letters*, *110*(8-9), 279-281. Dostupné z: https://doi.org/10.1016/j.ipl.2010.02.001

[21] Borg, I., & Groenen, P. (2003). Modern Multidimensional Scaling: Theory and Applications. Journal of Educational Measurement, 40(3), 277-280. Dostupné z: https://doi.org/10.1111/j.1745-3984.2003.tb01108.x

[22] KUMAR, Ashwani, Surinder PAL SINGH a Nitin ARORA. A New Technique for Finding Min-cut Tree. *International Journal of Computer Applications* [online]. 2013, **69**(20), 1-7 [cit. 2021-04-22]. ISSN 09758887. Dostupné z: doi: 10.5120/12084-9170s

[23] Path Finder: An Artificial Intelligence Based Shortest Path. *International Journal of Recent Technology and Engineering* [online]. 2019, **8**(4), 5177-5181 [cit. 2021-04-22]. ISSN 2277-3878. Dostupné z: doi: 10.35940/ijrte.D7380.118419

[24] Arora, N., Tamta, V. K., and Kumar, S. 2012. Modified Non-RecursiveAlgorithm for Reconstructing a BinaryTree International J. of Computer Applications (IJCA). vol 43. No 10. 25-28

[25] APPLEGATE, David L. *The traveling salesman problem: a computational study*. Princeton: Princeton University Press, 2006. ISBN 9780691129938.

[26] ARORA, Nitin, Somansh GARG, Varad SANT a Rohit GOYAL. Automated Optimum Route Generator and Data Analyzer. *International Journal of Computer Applications* [online]. 2019, **181**(48), 17-21 [cit. 2021-04-22]. ISSN 09758887. Dostupné z: doi: 10.5120/ijca2019918654

[27] CORMEN, Thomas H. *Introduction to algorithms*. 3rd ed. Cambridge: MIT Press, c2009. ISBN 978-0262033848.

[28] WINSTON, Patrick Henry. *Artificial intelligence*. 3rd ed. Reading, Mass.: Addison-Wesley Publishing Company, c1992. ISBN 978-0201533774.

[29] TRAVIS, Jeffrey a Jim KRING. *LabVIEW for everyone: graphical programming made easy and fun*. 3rd ed. Upper Saddle River: Prentice Hall, 2006. ISBN 0131856723.

[30] TANENBAUM, Andrew S. a Herbert BOS. *Modern operating systems*. Fourth edition. Boston: Pearson, [2014]. Global. ISBN 1292061421

Seznam příloh

Adresář elektronické přílohy obsahuje několik podadresářů:

A. 2021_GON0072_BP_priloha

    - Aplikace v LabVIEW

B. 2021_GON0072_BP_priloha

    - ukázka průběhů algoritmů na mapě 1

        1. DFS_průběh
        2. BFS_průběh
        3. Astar_výsledek